1991

An Object-oriented drawing package in smalltalk/v

Sigrid E. Mortensen

Rochester Institute of Technology
School of Computer Science and Technology

An Object-Oriented Drawing Package in Smalltalk/V

by Sigrid E. Mortensen

Thesis, submitted to

The Faculty of Computer Science and Technology

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Approved by:

Professor Michael J. Lutz

Robert T. Gayvert

Dr. James E. Heliotis

An Object-Oriented Drawing Package in Smalltalk/V

I, Sigrid E. Mortensen,

hereby deny permission for the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part.

Sigrid E. Mortensen 5/17/91

Acknowledgements

First, I would like to thank Rob Gayvert -- who has alternately taken on the roles of teacher, employer, mentor, advisor and friend -- for the vision and inspiration that never left me short of ideas; for his uncompromising scrutiny which forced me to justify all of my design decisions; and (perhaps most valuable) for his time, without which I would never have become so adept at Smalltalk.

I would like to thank Mike Lutz, both for keeping me from attempting an undoable thesis, and for providing the focus for this one. I also benefited greatly from his confidence in both my concept and my writing -- his enthusiasm for these continues to fuel my own, so that publishing seems well within reach.

Thanks also to Dr. Jim Heliotis for introducing me (and a host of others) to object-oriented languages in a way that made them so interesting, I think I'll never want to program in anything else.

Finally, special thanks go to James Tracy Burdick. His professional interest in my subject, and his finely-tuned intuition regarding all aspects of user interface design, made him an ideal sounding board for my ideas. Without his personal support, though, and without his encouragement for how and when to stop writing, this thesis would neither have been as quickly completed, nor as "short" as it turned out to be.

Abstract

Graphics creation applications tend to fall into two categories: bit-mapped paint packages, and object-oriented drawing packages. Although each interface has its own unique advantages, few vendors have attempted to integrate the two into a single package. Those who have tried have, in fact, poor integration both from the user's perspective and in the underlying mathematical model.

In this thesis, I have addressed the issue of integrating bit-mapped and object-oriented interfaces by creating an object-oriented graphics package which provides the user with a consistent interface for creating and manipulating both graphical objects and bit-mapped graphics. The consistency of the interface was facilitated by the consistency of the design, the underlying geometric model, and the implementation, all of which are themselves object-oriented. The thesis is written in Smalltalk/V for the Macintosh* .

While the solution for this integration was not derived overnight, the use of object-oriented design principles sped the development of a complex graphical user interface, while providing fresh insight into the problem of representing bit-mapped objects. Because Smalltalk enforces the notion that every element in the system is an object, the Smalltalk developer is forced to begin designing his solution purely in terms of objects. This mind-set allowed me to view the point as no other graphics package has presented it: as a unique graphical entity (just as it is in formal geometry) available to the user as a graphical tool. As a result, users of my package are able to enjoy the benefits of both bit-mapped and object-oriented editors without ever

* Smalltalk/V is a registered trademark of Digitalk, Inc.

abandoning an environment in which every graphical element is an object, in terms of both the interface and the underlying mathematical model.

Key Words and Phrases

Object-oriented languages
Object-oriented graphics (Drawing packages)
Bit-mapped graphics (Paint packages)
Smalltalk
Graphical User Interfaces

Computing Review Subject Codes

I.3.4: Graphical Utilities
D.3.2: Programming Languages
H.5.2: User Interfaces
I.3.6: Methodology and Techniques

An Object-Oriented Drawing Package in Smalltalk/V
Table Of Contents

Introduction

Graphics creation applications tend to fall into two categories: bit-mapped paint packages, and object-oriented drawing packages. Although each interface has its own unique advantages, few vendors have attempted to integrate the two into a single package. Those who have tried have, in fact, poor integration both from the user's perspective and in the underlying mathematical model.

In this thesis, I have addressed the issue of integrating bit-mapped and object-oriented interfaces by creating an object-oriented graphics package which provides the user with a consistent interface for creating and manipulating both graphical objects and bit-mapped graphics. The consistency of the interface was facilitated by the consistency of the design, the underlying geometric model, and the implementation, all of which are themselves object-oriented.

I have implemented my package in Smalltalk/V for the Macintosh. The reasons for choosing an object-oriented language in general, and Smalltalk/V in particular, for such a project are numerous. In the following section, I give a brief history of the object-oriented languages leading up to Smalltalk, then discuss object-oriented concepts and terminology, and explain in detail my own motivations for choosing Smalltalk/V.

In the section titled "Graphical Interfaces," I discuss the relative merits of both bit-mapped and object-oriented graphics packages, and follow with a discussion of object-oriented graphics as they have been traditionally used in general-purpose, event-driven graphical user interfaces (GUIs). Due to the complexity of programming GUIs, there is a growing application of object-oriented concepts to the creation of these interfaces. I discuss some of the

motivations for using object-oriented techniques for programming GUIs, as well as some of the drawbacks which still exist in the models used for GUI creation. I conclude this section with a comparison of the classical graphics standards (GKS and PHIGS) to object-oriented implementations in the building of object-oriented graphics packages.

Related work falls into four categories: object-oriented concepts used to simplify programming GUIs; object-oriented concepts for programming specialized graphics applications (like animation); object-oriented concepts used to create object-oriented drawing packages; and commercial drawing and painting packages written in any paradigm. For the last category, I outline the features of existing packages, and discuss some of the complaints leveled against the interfaces of these packages.

In the "Results" section, I describe in detail my own implementation. I have a running, two-dimensional, black-and-white drawing package which integrates both bit-mapped and object-oriented drawing creation and editing. In particular, I discuss some of the design decisions which led to the high level of interface consistency in the current version. Object-oriented design and implementation had a great impact here, and also helped overcome some of the traditionally cumbersome problems of object management which arise when programming object-oriented drawing packages in procedural or functional languages.

Future extensions are numerous. They range from simple classes of graphical objects which can be easily added to the system, to some rather complex features, to enhancements of some aspects of the user interface. These are discussed in the section on "Conclusions and Future Enhancements."

Object-Oriented Concepts

History: The Smalltalk Language

In 1971, Alan Kay began developing a programming environment he called Smalltalk. This development was two-fold -- developing the language, and developing a user interface to "match the human communication system to that of the computer." [Goldberg and Robson,1989, p. vii] Both the language and the interface would be object-oriented, although, as we will see, the use of the term "object-oriented" to describe them is coincidental; an object-oriented interface does not depend upon an object-oriented language.

Simula's Influence

Smalltalk was not the first object-oriented language. These languages have roots stretching back into the mid-1960's, when Simula-67 was created: a language whose use of classes not only simplified the simulation programs for which it was designed, but inspired its successors. Early Smalltalk was "heavily influenced by [Simula's] idea of classes as an organizing principle." [Deutsch, 1989, p. 74]. In 1976, PARC developers were again borrowing from Simula; Smalltalk-76 included Simula's new concept of inheritance. Smalltalk-76 also refined much of its earlier user interface, creating a class library of windows, scroll bars, menus, lists and text editors "all reusable through subclassing." [Deutsch, 1989, p. 77] [Thomas, 1989] [Wegner,1989]

Smalltalk-80

Smalltalk's own impact on the programming community was not immediately felt, in part, perhaps, because of its innovative graphical interface which took the initial spotlight, and partly because it ran only on expensive machines and was not specifically marketed toward a mass audience. ([Thomas, 1989] notes, however, that no other GUI allowed the user to modify

the underlying code.) In Xerox's 1979-80 development year, PARC began to make Smalltalk more portable by removing some machine dependent access, and restricting the fonts to conform to the ASCII character set. The resulting Smalltalk-80 also introduced the ideas of both code "blocks," and classes themselves as objects, which made the language completely consistent: there is nothing in Smalltalk which is not an object. Smalltalk/V, the implementation language used for this project, is a large subset of Smalltalk-80 put out by Digitalk Corporation, which has versions for both the IBM PC and the Macintosh. [Thomas, 1989] [Seiter, 1989] [Deutsch, 1989]

Terminology

There are now many object-oriented, object-based and class-based languages available. (For the subtle distinctions, see [Wegner,1989].) C++, Objective-C, Eiffel, and the LISP-based Flavors and CLOS are just a few examples. The terminology varies slightly -- a "class" in Smalltalk is a "flavor" in Flavors -- but the concepts are fairly consistent. Where there are differences, this paper will use the terminology of Smalltalk, specifically Smalltalk/V. Also, the browsers shown in the diagrams are modified Smalltalk/V browsers.

Objects, Classes and Instances

The basic building block of an object-oriented language is the *object*. Conceptually, objects are animate, autonomous entities, each with its own specific attributes (data) and functionality. Objects are organized into *classes* the way individual cats, for example, all belong to the class Cat; each cat is said to be an *instance* of the class Cat.* It is the class' description which defines the

* By Smalltalk convention, class names are capitalized, and instances are lower case.

behavior of the class' members. In this way, we can expect each cat to behave in approximately the same manner as every other cat.

Instance Variables

Besides behavior, the class description also includes the *instance variables* to potentially hold the attributes of each instance. Each instance has its own copy of these instance variables, which can be filled with its own particular attributes. The values of these variables within the instance may, in turn, affect the behavior of that particular instance. The class Cat, then, may include in its description instance variables for name, color, weight, age, and health.

Figure 4 shows a partial Smalltalk browser on the hypothetical class Cat. The class name is in the upper left pane and the class description is in the lower pane. The second line, "instance variables," shows this list of attributes. One instance of Cat, then, may be Felix, orange, 9 pounds, 2 years old and healthy. Felix's behavior might differ from that of Puff (white, 12 lbs., 13 years old and weak). When asked to show their pictures on the screen, their colors will obviously be different. When made to respond to the sound of a can opener, Felix may move faster. That they are able to show themselves on the screen or react to can openers are behaviors defined by the class.

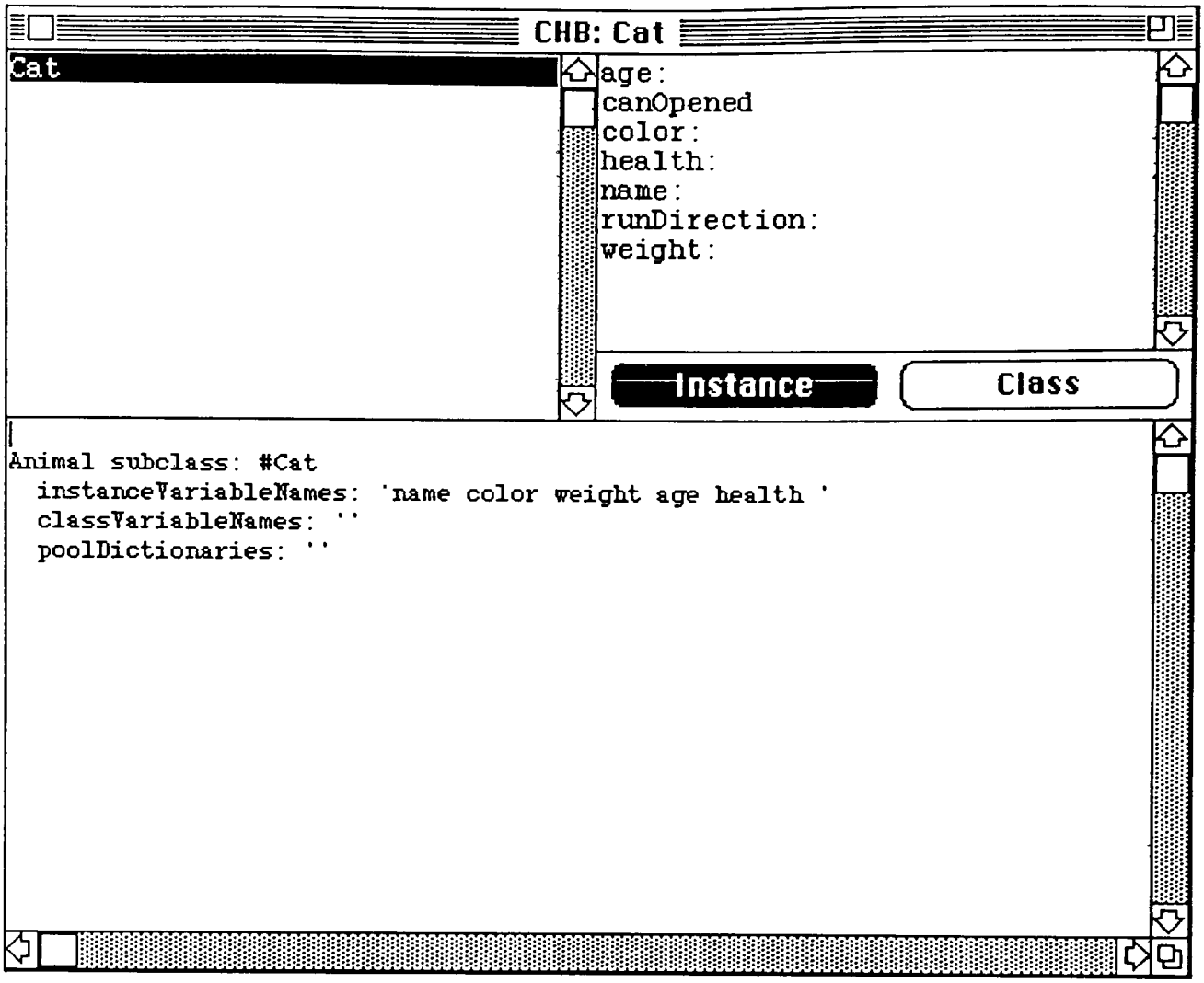


Figure 4 -- Browser on Cat class

Messages

Objects communicate with each other through the passing of *messages*. In the upper right-hand pane of figure 4 is a list of some of the messages to which a cat may react. Some of these (name:, color:, etc) are used to set the instance variables. There are also messages for initiating actions in or eliciting responses from the cat instance. An instance of another class, CatOwner, may, in the process of feeding the cats, send each cat the message

```
cat canOpened.
```

and wait to see how it reacts. Such a message may return information back to the sender (such as how hungry the cat feels), or it may initiate some action in the receiver (or it could do both). Such an action could, in turn, cause the receiver (the cat) to send further messages to itself, such as

```
self runDirection: towardFood.
self eat.
```

where **self** is a pseudo-instance variable implicitly defined for each class. The above syntax is that of Smalltalk: the receiver object is written first, then the message to that receiver.

Methods

How a class of objects reacts to a given message is defined by the *method* associated with that message. Every message has a corresponding method which tells how the message should be carried out. An object's messages are the interface between it and any other object that wants to communicate with it; and the methods are the underlying code. The cat owner knows, then, that each cat will react somehow to the message **canOpened**. It is not clear to the cat owner, though, whether a heavier cat will run faster or slower toward food; that decision is left to the discretion of whoever created the Cat class and its methods. If that programmer later decides that health should also play a role in speed, he may change the method by which a cat reacts to **canOpened**,

while still retaining the same interface (reacting to **canOpened**) and functionality (eventually eating). This allows the cat to be a black box, with observable inputs and outputs, but hidden "thought processes." This is information hiding in the sense that the methodology employed to carry out a message is hidden.

Public vs. Private

Some object-oriented languages have the concept of public and private messages and data. In these languages, **canOpened** of the above example would be considered a public message: one which any object can send to a cat; but both running and eating are private: only a cat can tell itself to perform these actions, no one else can. In Smalltalk, all messages are officially public and all data is private. A programmer may include comments in a message to indicate that the message was intended to be private, but anyone actually has access to that message; the privacy is not enforced by the language. On the other hand, if an object chooses either to reveal its data, or to allow another object to set the value of one of its instance variables, the object must explicitly include a message/method pair to return or set it.

Message Selectors

The above examples also illustrate that messages may have zero or more arguments. The message **canOpened** had no arguments, while the message **runDirection: towardFood** had one argument. In Smalltalk, arguments follow colons. A message with the colons but without the arguments makes up the *message pattern* or *selector* (e.g. **runDirection:**). A message is paired with its method by matching the message pattern; the method may have any variable name as a place holder for the input parameter.

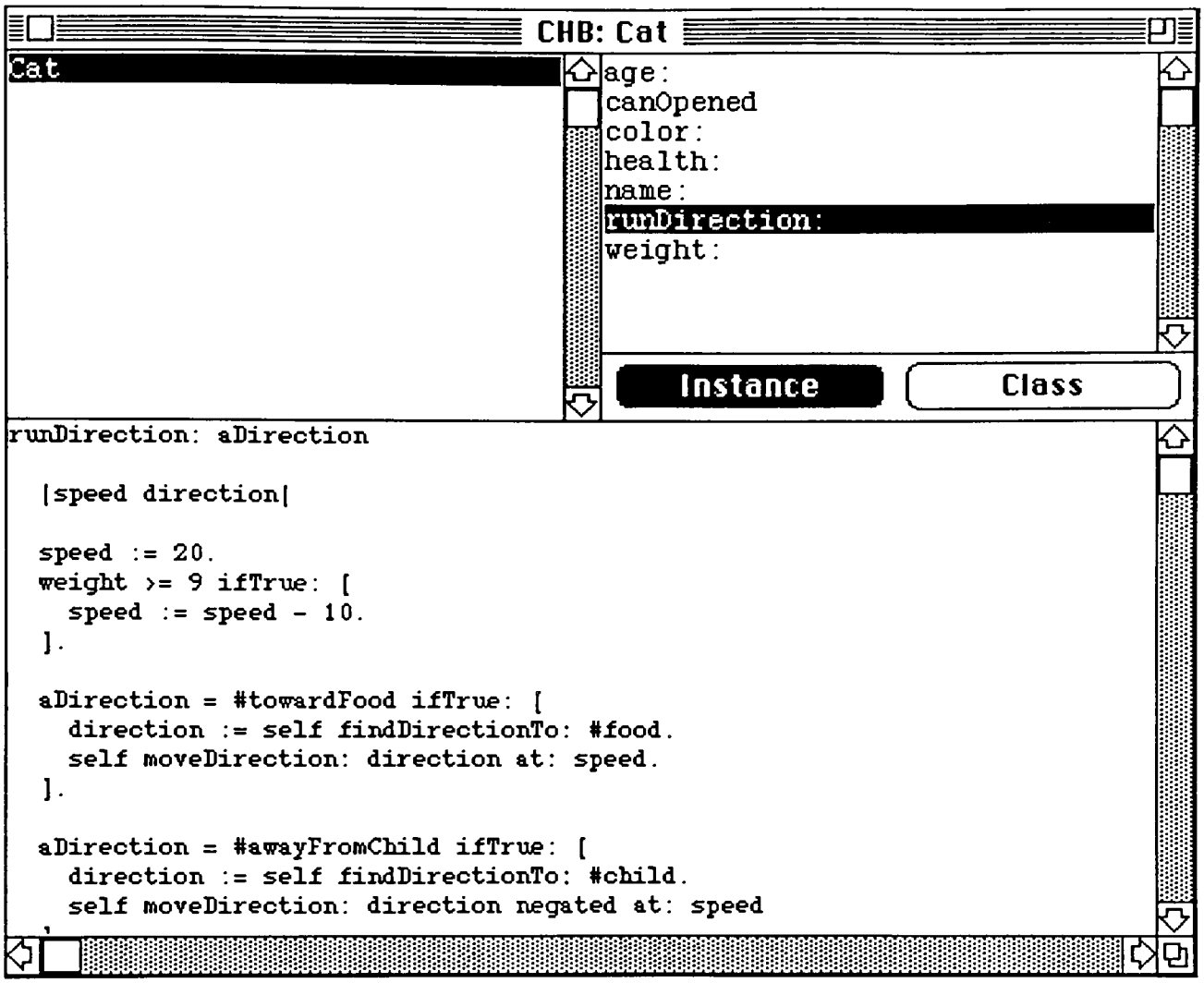


Figure 5 -- Cat Browser, method selected

Figure 5 shows the Cat class with one of the message selectors (**runDirection:**) in the upper right hand list highlighted. This allows the code for the method to be visible to the programmer in the lower pane. When the message **runDirection: towardFood** is sent to a cat instance, **towardFood** gets associated with **aDirection** at run-time. (In reaction to a different message, the cat may send itself the message **runDirection: awayFromChild**, but the method would remain the same.) There are no output parameters in Smalltalk. To send information back to the

sender, an (arbitrarily complex) object may be returned via a return statement.

Inheritance

Smalltalk also supports the concept of *inheritance*, i.e. one class of objects may be defined in terms of another, enabling it to derive some of its attributes and behavior from the pre-existing class. The new class, the child, may be similar to its parent class in some respects, but refine, expand, or restrict the capabilities of that parent.

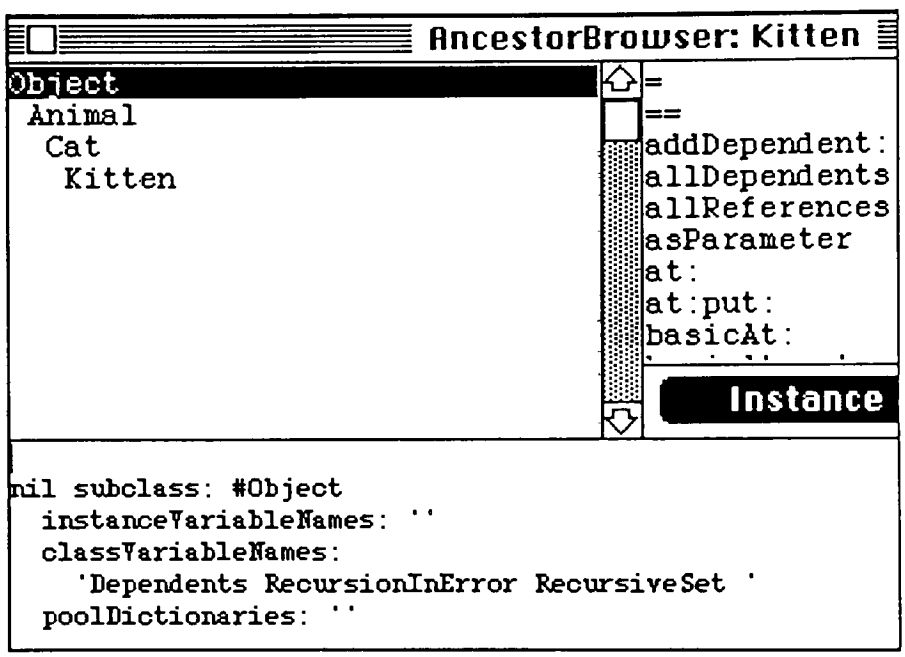


Figure 6a -- Object class

The Cat class, then, may be a child (or *subclass*) of a pre-existing class Animal, getting its abilities to eat, breath and move from its parent (or *superclass*). It may inherit eating and breathing without alteration; redefine moving to accommodate all four legs and the abilities to run, walk, stalk, and pounce; and add the abilities to purr and react to can openers. Cat may then be further subclassed by Kitten which may add a playfulness attribute. Kitten inherits all the capabilities of Cat, Animal, and the parent class of all classes:

the class Object. (Object provides rudimentary functionality such as the abilities to create new instances or print an instance.) Figures 6a-c show the class descriptions and hierarchy of each of these classes. Each level of indentation indicates a level of inheritance.

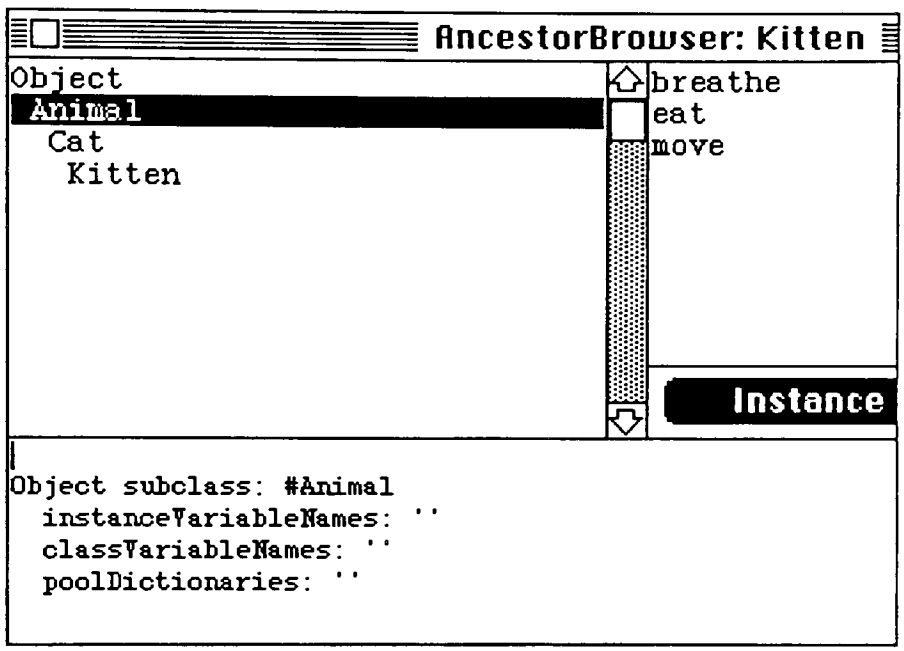


Figure 6b -- Animal class

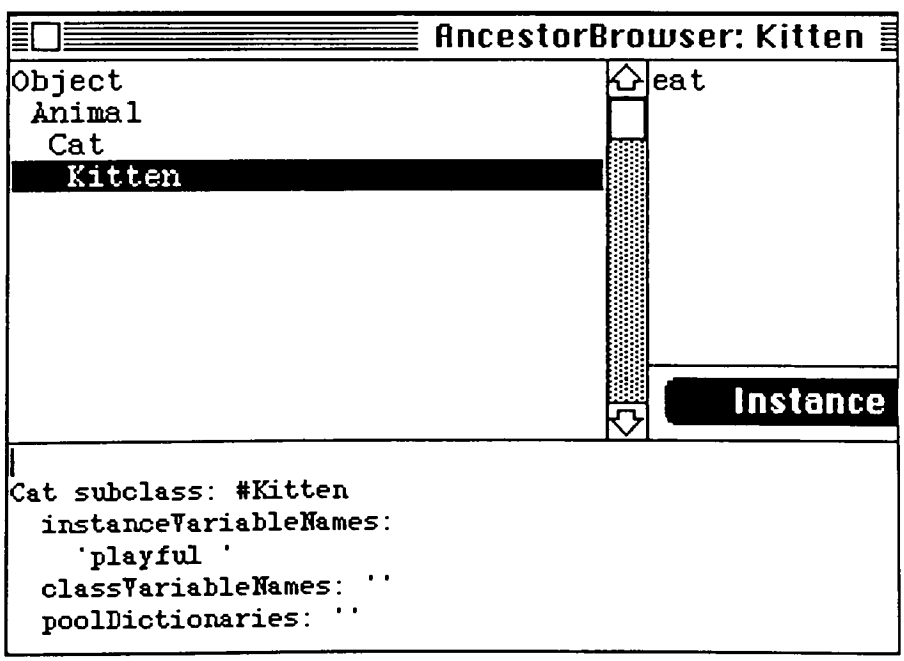


Figure 6c -- Kitten class

The search for methods to match messages always begins in the class of the receiver, then moves up the hierarchy. When a kitten sends itself the message to eat, then, the method will first be sought in the Kitten class, then the Cat class, and finally the Animal class, where, in this case, it would be found. If the message were not found in any of the parents (including Object), an error would be generated. It is also possible for a method to invoke its parent's method with another psuedo-instance variable, **super**. Thus, if kittens always nap after they eat, the **eat** method in the Kitten class might look like:

```
super eat.
self sleep.
```

In these examples, the Animal class could be considered an *abstract class*, since there are no objects which are instances solely of Animal, without further being defined as Cats, Dogs, etc. Animal defines abstract animal behaviors, leaving the specific behavioral refinements to its subclasses.

Multiple Inheritance

Some object-oriented languages (Smalltalk not included) further provide the concept of *multiple inheritance* where, instead of a single parent, a class may have one or more parents. A HouseCat class, then, might inherit from both Cat and HomeFurnishing, from whence it derives the need to come in out of the rain. Whereas single inheritance can be represented by a simple tree structure, multiple inheritance requires an acyclic directed graph. Multiple inheritance is further complicated by the need to specify in which parent each inherited behavior is implemented, since there is no longer a single **super**.

Promised Benefits

An obvious question this thesis attempts to answer is: why would (or should) someone use an object-oriented programming language to create an object-oriented graphics package when such packages have already been successfully written in other paradigms? In particular, what motivation is there for using Smalltalk for such a venture?

One of the first and most obvious reasons for writing this package in Smalltalk/V is that this is the current development and implementation language for my division at the RIT Research Corporation, and that the package for this thesis is a small part of a large, integrated software product we are developing there.

Buzz Words

Further justification is found throughout the literature for using object-oriented languages for *any* application. Here a slew of buzz words and phrases crop up including: specialization, generalization, abstraction, incremental evolution, incremental modification, prototyping, code reuse, polymorphism, integration, consistency, encapsulation, programmer development environment, modeling the real world, and programmer efficiency. All of these apply to some aspect of object-oriented programming and all are valid reasons for using object-oriented languages in general (and Smalltalk in particular) for program development.

Specialization, Generalization, and Abstraction

Most of the elegance of object-oriented programming comes from inheritance. It is the existence of parent classes which allows the programmer to generalize his solution, particularly through the use of abstract classes. The subclasses then specialize the behaviors of the parent class. One programmer may then hone his own or someone else's earlier work

to fit a particular problem. In practice, abstract classes often grow out of vaguely-formed subclasses when the programmer recognizes the subclasses' like behaviors. With practice, the programmer learns to recognize these similarities early, and incorporates them into the design of his class structure. [Wegner,1989] [Wisskirchen and Rome, 1988] [Fuchsberger and Krasemann, 1990]

Code Reuse, Encapsulation, Programmer Efficiency

The benefit of code reuse also applies to inheritance, since any code inherited without modification can be reused; but code reuse would also be possible even if inheritance were not available. Every time a programmer creates a class within the system, that class and all its methods have the potential to become available to other programmers. Therefore, if someone has already written a Stack class, that class is neatly packaged (encapsulated) for use by other programmers, and no one else has to write algorithms for pushing and popping. [Wisskirchen and Rome, 1988] [Thomas, 1989] [Dodani et. al., 1989]

Code reuse naturally leads to programmer efficiency, since all programmers are not coding the same things repeatedly. In addition, Smalltalk's automatic garbage collection contributes to programmer efficiency, since the programmer need not concern himself with allocating and deallocating storage. Smalltalk's rich set of classes also means that Smalltalk code is compact. "Smalltalk source code is the equivalent of five or six statements in Pascal or C." [Seiter, 1989, p. 192] "Studies have shown significant reductions in both development time (by 75%) and size of source code (by 90%) when object-oriented techniques were used." [Yankelovich et. al., 1988, p. 13] [Kramer,1984] [Urlocker, 1990] [Deutsch, 1989]

Integration and Consistency

The availability of code from pre-existing classes also lends both consistency and ease-of-integration to the programming process. The library of existing code means that a programmer can expect all the basic classes to behave consistently (respond to the same subset of messages) from one application to the next. This, in turn, makes it easier to link the classes created by separate programmers into a coherent whole. Also, the ways in which objects relate to each other in the Smalltalk kernel often set a precedent for future applications. Smalltalk's model-view-controller paradigm (see page 33) is an example of a standard for programming user interfaces that, while not an explicit part of the language, is used in many Smalltalk programs. [Mohamed, 1987] [Thomas, 1989]

Polymorphism

Polymorphism (meaning "many forms") refers to a method's ability to accept any data type for an argument. This is made possible by Smalltalk's run-time binding. Methods are compiled, but the compiler checks only syntax and the existence of classes referenced. Data types of instance variables, as well as the availability of methods to match the messages sent, need not be resolved until the code is run. Further, the encapsulation of each class means that different classes can respond to the same messages with no conflict. Therefore, if both Arrays and OrderedCollections respond to the message **at: anIndex**, either can be the argument for a method which sends this message to its argument. It is also possible, in Smalltalk, to react according to the class of an object with messages such as:

```
(anObject isKindOf: ClassName) ifTrue: [...
```

or to find out if a message is an appropriate one to send to an object with:

```
(anObject respondsTo: aMessage) ifTrue: [...
```

This allows the programmer the flexibility of reacting differently depending on the argument's class. [Thomas, 1989] [Thompson, 1989] [Meyers, 1988]

Incremental Evolution, Prototyping and the Development Environment

Smalltalk's programming environment provides, as well as the direct manipulation interface with which it was developed, access, both visual and actual, to the underlying code of all classes, even those classes which are a part of the Smalltalk kernel. It also includes workspaces in which to quickly explore simple algorithms; interactive debuggers; and the ability to find all the senders or implementors of a given message. Incremental modification, incremental evolution, and rapid prototyping are made possible just as much through this environment as they are through inheritance. While inheritance makes incremental development possible, quick compilation and the ability to quickly evaluate an expression means that results are readily available for scrutiny and modification. "In contrast to classical programming languages (coding, compiling, test), programming in object-oriented environments is based on the heavy use, modification and reuse of available parts." [Wisskirchen and Rome, 1988, p. 21] [Wegner,1989] [Kramer,1984] [Dodani et. al., 1989]

Modeling the Real World

A final argument for using object-oriented design for general purpose programming is that, since the real world is made up of objects, such languages more closely model the world than do, say, procedural, functional or linear languages. To date, there is little evidence that people think more efficiently or naturally in terms of objects; but objects, both concrete and abstract, provide a convenient way of grouping together data and behavior, which in itself is a powerful tool for managing the complexity of any large project. Smalltalk, in which everything is an object -- even classes and pieces

of code -- completely indoctrinates the programmer into this object-oriented view of the world. Once the programmer has come to think only in terms of objects, program *design* takes on its own consistency, where the programmer can generally ask, "What are the objects?" "What other objects make up these objects?" and "What should these objects do?" [Meyers, 1988] [Dodani et. al., 1989] [Haaland and Thomas, 1989] [Tazelaar, 1989]

Graphical Interfaces

There are a number of graphics packages available on the software market today, ranging in scale from small paint programs on personal computers to large CADD packages for architects and engineers. Their tools extend from the barest minimum needed to create a simple drawing, to a wide range of features for creating and manipulating complex graphical images. Some packages include color; some even take the user into the third dimension. In spite of their differences, however, they can be grouped into four general categories:

- paint programs using bit-mapped graphics;
- presentation graphics for creating charts, slide shows, and reports (usually from spreadsheet data);
- object-oriented (also called object-based or vector-based) illustration packages;
- CADD packages. [Kerlow, 1989]

Since CADD packages also tend to be object-oriented, the three of these categories which allow for the creation of original drawings or designs (paint programs, object-oriented, and CADD packages) can further be lumped into two distinct classifications: bit-mapped and object-oriented. In spite of the proliferation of graphics packages, it is rare to find an application which attempts to combine characteristics of both bit-mapped and object-oriented

graphics. [Cox and Caldeway, 1987] [Fenton, 1989a] [Kirsh,1988] [McKinstry, 1989]

Bit-Mapped Graphics

A bitmap is a graphical representation in which each picture element (or "pixel") takes on a numeric value. In a black-and-white system, for example, the bitmap is binary: a black pixel is represented as a 1, and a white pixel as a 0. Ultimately, all black and white images (even object-oriented graphical images) must appear to the user as a bitmap. [Heid, 1989]

Characteristics

In a bit-mapped graphics package, the user creates images as if by stroking paint onto a canvas, where the underlying bitmap is the canvas. Because an individual bit can only have one value, the paint is opaque. In a color system, for example, if a blue square is drawn on top of an orange square, the blue square obliterates the orange square. The orange square cannot be selected, moved, or manipulated because it no longer exists. [Jones, 1989] [Roth, 1989]

Bit-mapped graphics have the following characteristics:

> "1) each pixel is a singular unit,
> 2) the interface recognizes only pixels and not graphic primitives per se, and
> 3) portions of a graphic or any pixel(s) may be manipulated independently." [Mohageg, 1989, p. 385]

Advantages

Because of their pixel-based orientation, bit-mapped graphics packages have a few advantages. They are well-suited to finely-detailed graphics where manipulation of individual pixels is important. This includes subtle shading techniques, smudging effects, graduated fills, and (perhaps most coveted by users of object-oriented graphics packages) the ability to erase part of the drawing. [Mohageg, 1989] [Pogue, 1989] [Busch, 1986]

Problems

Resolution Dependent

Unfortunately, bit-mapped graphics packages are also beset by a number of inherent problems. First, since a bit-mapped graphics package has no knowledge of graphical elements larger than each individual pixel, the graphics are resolution dependent. A graphic created on a 72 dots-per-inch display screen, even when printed on a 300 dpi laser printer, will have a resolution no finer than 72 dpi. This is also noticeable when an image is enlarged (or "stretched") on the screen. Each square pixel becomes a square of four or more pixels, so curves and angled lines that once looked relatively smooth become jagged, like stair-steps. In traditional bit-mapped packages, even text is part of the bitmap, making it also subject to jagged lines upon enlargement. The newer packages have attempted to overcome this problem with the inclusion of 300 dpi editing (with all the tedium that implies) and real text. [Kerlow, 1989] [Pogue, 1989] [Roth, 1989]

Selection Box

An even more frustrating problem, from the user's point of view, concerns the editing of a bit-mapped picture. Although these packages are well suited to the manipulation of individual pixels, the user must do considerably more work to manipulate a larger part of the picture. It is not possible to simply pick up a circle and move it; instead, the circle must be cut out of the bitmap. Usually this is done with a tool popularly known as the selection box. After choosing this tool, the user is able to click the mouse at one corner of the area to be cut, then drag the mouse while a rubber-banding rectangle follows the cursor. When the mouse button is finally released, the resulting rectangle is highlighted to let the user know that it is now manipulable. It can be moved to another portion of the screen, deleted, stretched or reduced and, in some

packages, rotated, skewed, or flipped horizontally or vertically. Note that this process is like cutting a hole in the canvas: no previously applied paint will appear under what was moved; instead, the blank background color (usually white) will remain.

Lasso

While the selection box is an improvement over individual pixel manipulation, it exhibits a limitation when the portion of the graphic to be moved is diagonally adjacent to another graphic. Figure 1 shows a square that is supposed to stay where it is, and a circle that the user wishes to move with the (dotted-line) selection box. Notice that part of the square will be cut along with the circle, making it necessary to 1) go back and fill in the parts of the square that were supposed to remain and 2) remove the stow-away square parts from the circle's ultimate destination.

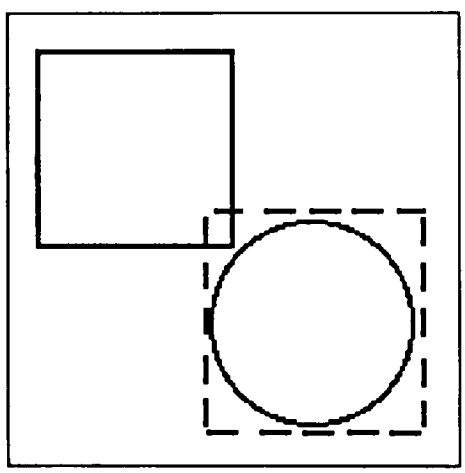


Figure 1 -- Selection Box

Most pixel-based packages offer a partial solution to this problem with another cutting tool known as the lasso. Like a free-drawing or pencil tool, the lasso follows all the pixels in the path of the cursor as long as the mouse button is pressed. When the mouse button is released, the lasso will complete its circuit (if the user has not already done so) and shrink to enclose tightly

any pixels that are not part of the background. The bits that make up the circle in the previous example can now be manipulated independently of any neighboring graphics. [Busch, 1986]

Overlapping

The lasso is still not a complete solution, however. When graphics are close together, for example, it requires a steady hand to maneuver the tool cleanly through tight gaps. This is difficult but not impossible. The ultimate editing dilemma in a pixel-based package is illustrated in figure 2. When a circle and square overlap, no amount of lassoing will grab only the circle. The user must laboriously cut out first the lower right-hand part of the circle and move it; then, assuming he wants to keep the circle, cut out the upper left-hand part, taking care to place it from the lower part a distance the width of the square's border (because as soon as he releases it and clicks elsewhere, it again becomes part of the bitmap); then fill in the missing bits. (See figure 3.) In an even more complex drawing (with, for example, a fill pattern inside the square) this editing becomes very tedious indeed. [Jones, 1989] [Busch, 1986]

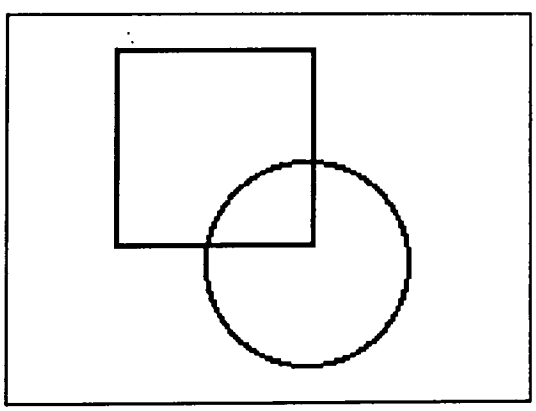


Figure 2 -- Overlapping Objects

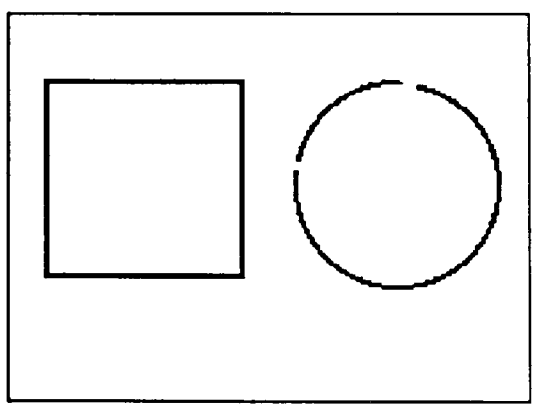


Figure 3 -- After Move

Object-Oriented Graphics

Better editing solutions are made possible by the very nature of the object-oriented graphics packages. Drawing with these packages has been likened to

creating a collage of either construction-paper cut-outs, or a collection of differently shaped rubber-bands to create a picture, instead of painting on a canvas. Of the two, the rubber-band analogy is more accurate, since graphical objects can be transparent, and can be freely selected and resized. [Busch, 1986] [Heid, 1989] [Kerlow, 1989]

In an object-oriented package, graphics are not stored as bitmaps; in fact, there is no recognition of individual pixels. Instead of pixels as the finest granularity, object-based packages use mathematically defined shapes as their smallest editable units. Lines, for example, are defined as a beginning point and an ending point; rectangles as an upper left-hand corner and a lower right-hand corner. Instead of remembering all the bits in the bitmap that a particular graphic covered when it was created, these programs store the attributes of the graphic. "Draw a circle and the program makes notations in memory that you drew a circle with radius *a* at position *b* with a border width of *c* and fill pattern of *d*." [Heid, 1989, p. 198] Because these graphics are not tied to a bitmap, they are resolution and device independent. When the user enlarges a circle with a given border width, only the radius of the circle changes, not the border width; if he wants a different border width, he may change that with a menu command. [Kerlow, 1989] [Roth, 1989] [Burns and Venit,1987] [Mohageg, 1989]

Selection of Objects, Grouping

Because each graphical object is an individual entity in an object-oriented package, editing becomes a simple matter of selecting (usually with a single mouse-click) the object to be manipulated, and moving, stretching or rotating that object. (The selection is usually apparent to the user with some sort of highlighting: the object is a different color, flashes, has a dotted border or, most frequently, displays "handles:" small black squares at regular intervals

around the perimeter of the object -- see figure B-10 in Appendix B.) If objects overlap one another, the objects which are underneath the selected object will stay in place, undisturbed by the movement of their neighbor. Most applications also include the ability to "group" objects--select more than one object at a time, and apply the same transformation to all the objects in the group. Unlike object selection in the bit-mapped interface, these objects do not have to be touching or even near each other for the multiple selection to take place.

Problem, No Erasing

In the object-oriented packages, then, there is no exacting tracing around objects, no piecing together of objects, and no difficulty with overlapping objects. There is also no erasing. Bit-mapped packages usually have an eraser tool with which the user can wipe out larger areas of the drawing than he could by manipulating individual pixels. But the eraser is a tool noticeably absent in object-oriented packages. This is tied to both the object-oriented philosophy: that "each graphic primitive is a singular unit...and portions of graphics are not editable" [Mohageg, 1989, p. 385], and to the underlying mathematics that describe the images. Although there is a clean and simple mathematical formula for defining a rectangle, for example, it is more complicated to define a rectangle with part of one side erased. If it is a filled rectangle with a path erased through it, the mathematics are harder still. It is, however, the lack of an erasing feature that draws the most complaints against object-oriented drawing packages. [Busch, 1986] [Mohageg, 1989]

Study Comparing the Two Interfaces

The Human Factors Engineering Laboratory at the Virginia Polytechnic Institute performed a study to compare the differences in both performance and preferences between bit-mapped and object-oriented graphics interfaces.

They queried twenty expert users of the interfaces to determine the most frequently performed tasks during normal usage; then tested 24 subjects, half of whom were novice users, and half experienced. The researchers concluded that the object-oriented interface was superior to the bit-mapped in terms of reduced learning time, fewer aborted attempts, fewer errors, and faster project completion time. Furthermore, users overwhelmingly preferred the object-oriented interface. [Mohageg, 1989]

Performance

Although the original creation of graphical objects such as circles and rectangles was no faster on the object-oriented interface, graphic manipulations such as scaling, rotating and moving favored this interface. "The object-oriented interface had significantly smaller task completion times than the bit-mapped interface for 23 of 29 tasks." [Mohageg, 1989, p. 387] This may have affected the time it took to learn the respective interfaces, since even after task completion times had stabilized for novice users of both interfaces, they were shorter for the object-oriented interface. Also significant for learning time was that while task completion times stabilized early for the object-oriented interface (as early as the second trial), they "continued to decline significantly" [Mohageg, 1989, p. 388] for the bit-mapped interface.

The number of repeated or aborted attempts also affected task completion times. In 16 of the 29 tasks, the bit-mapped interface showed a significantly higher frequency of repeated and aborted attempts: 58 vs. 310. Further, there were fewer errors, defined as deviations in size or position of individual graphics, in the object-oriented interface. "The interface type had an effect on size deviation in 10 of the 20 possible errors, and all of these tasks indicated significantly smaller deviation for the object-oriented interface. The 10 errors

occurred for graphic manipulation, rather than drawing, tasks.... Additionally, there were significantly fewer total positioning errors for the object-oriented interface (9) than for the bit-mapped interface (32)." [Mohageg, 1989, p. 387]

Preferences

The final part of the study measured the subjects' preferences for the two interfaces. All three preference measures: absolute, comparative, and overall, favored the object-oriented interface. "Twenty-four significant differences were obtained, of which 23 were in favor of the object-oriented interface.... No differences were found in drawing tasks. Additionally 83.3 per cent of subjects [20 out of 24] selected the object-oriented interface as the overall preferred interface for performing these tasks." [Mohageg, 1989, p. 388] The one feature that subjects liked in the bit-mapped interface was the ability to erase. Of the 24 differences, "erasing graphics was the only action preferred on the bit-mapped interface." [Mohageg, 1989, p. 388]

Performance and Preference Conclusions

The author explained the overall superiority of the object-oriented interface by citing the (previously-illustrated) difficulty of selection in the bit-mapped interface.

> [In the bit-mapped interface,] when using a selection box to surround the graphic of interest, a user may accidentally include portions of other graphics, at which point he/she must start the selection task anew. In contrast, to perform the same operation, object-oriented users simply point to any part of the graphic with the cursor. As a result, graphic selection in the bit-mapped interface tends to be a more time consuming and error prone operation. [Mohageg, 1989, p 388]

The coarser granularity (objects as the smallest units instead of pixels), and the ability to overlay graphics and still have them retain their identity as objects were also given as possible explanations:

> In the bit-mapped interface once graphics are overlapped or superimposed the pixels of the graphics are no longer

> distinguishable from each other. This factor also contributed to the significantly longer task completion times and higher error rates for the bit-mapped interface. [Mohageg, 1989, p 388]

The author concluded that for most applications, the object-oriented interface is superior. The one exception is in a drawing or painting situation, such as an art application, where manipulation of individual pixels or parts of objects is important. Since the object-oriented interface is not designed for such manipulations, it is predictably not well suited to them. The author further states that such individual pixel manipulations are rare in most drawing situations. [Mohageg, 1989]

Object-Oriented Graphics for Interfaces

Object-oriented graphics were originally developed not for drawing programs, but for user interfaces. Called graphical user interfaces (GUIs), or direct manipulation interfaces (DMIs) for the user's ability to interactively affect the size and position of the visual components, such interfaces are characterized by bit-mapped display screens, icons, scrollable windows, pull-down or pop-up menus, and some pointing device such as a light pen or mouse. These interfaces are also generally considered more user-friendly than purely textual interfaces. Users report positive feelings of confidence, control, and a willingness to explore the features of the applications which use them. [Jones, 1989] [Shneiderman, 1987]

Some of the earliest development of direct manipulation interfaces was begun at Xerox's Palo Alto Research Center (PARC) in the early 1970's. At that time, PARC's researchers attempted to predict what technology would be available ten years into the future and began the software development for products that would run on that future hardware. They realized the importance of the interface in the usability of a system, to the extent that they

began designing such interfaces even before the hardware that could support them was built. [Smith et. al., 1982] [Kay,1980] [Goldberg and Robson,1989]

Smalltalk Interface

As mentioned earlier, Smalltalk was designed to be an entire programming environment with not only an object-oriented language, but also an object-oriented user interface. This interface was influenced by the "turtle graphics" of MIT's LOGO; by the DMI of the FLEX machine, on which Alan Kay wrote his Ph.D. thesis; and by Ivan Sutherland's *Sketchpad*, a "general purpose graphics system for interactive creation and editing of pictures on a graphics display." [Lorensen et. al., 1987, p. 2] Kay had to convince his employers at PARC to invest in bit-mapped displays for the project, which, at the time, were "considered expensive and arcane." [Deutsch, 1989, p. 75] It was this step, however, that allowed Kay and his co-workers to develop bit-mapped text in a variety of fonts and sizes, multiple windows, and the Bit Block Transfers (or BitBlts) which make possible the manipulation of large amounts of graphical data in real time. [Deutsch, 1989]

Star Interface

The earliest Smalltalk system (for it has had, so far, at least four incarnations) was developed on Xerox's Alto computer. An immediate successor to this machine was the Star, where the developers also focused intently on the interface. They made a serious attempt to model what would be familiar to the user as much as possible, for which they relied heavily on icons. Since it was designed for an office system, there were icons for in- and out- baskets, file folders, printers, and documents. They also borrowed some object-oriented programming concepts for grouping similar elements of the interface together, which lent a high level of consistency to the interface. [Smith et. al., 1982]

Neither machine was a commercial success, however. It was when a fellow from Apple Computer by the name of Steve Jobs saw Smalltalk demonstrated at PARC that the seeds for the Macintosh computer were sewn. The Macintosh (which grew out of the less successful Lisa computer) became the first computer targeted toward the home market to include an DMI. In fact, the Macintosh interface is exclusively graphical; even at its root level, the user sees only icons and pull-down menus. It has since set an industry standard. Many machines offer DMIs as at least a part of the interface. Microsoft offers Windows for IBM-compatible machines; Sun and Amiga include graphical interfaces; LISP machines offer the object-oriented language Flavors which includes a DMI; and both the Atari and NeXT home computers have built-in interfaces with a look and feel similar to that of the Macintosh. [Seiter, 1989] [Cox and Caldeway, 1987] [Giguere, 1990] [Texas Instruments, 1987b]

Direct manipulation interfaces are "object-oriented" in an intuitive sense. Windows, icons, even the "thumbs" in scroll bars are all objects in the same way that the graphical objects in an object-oriented drawing package are. These objects can be picked up and moved, and whatever objects existed beneath them will remain where they were, unaffected. Windows can also be stretched or shrunk, and still retain their identity as windows. In spite of the common terminology, however, object-oriented graphics (interfaces included) are not dependent upon object-oriented languages. It does not require an object-oriented language to create either an object-oriented interface or an object-oriented drawing package; most of the existing ones are written in Pascal, assembler code, and C. [Wisskirchen and Rome, 1988]

Programming Graphical User Interfaces

I have already discussed several reasons for choosing to program in an object-oriented language. While these arguments are certainly valid for using

object-oriented languages for any domain, they are even more compelling when applied to programming graphical user interfaces. The consistency and ease-of-use of these interfaces is making them increasingly popular, a trend that places a demand on software developers to either offer GUIs or watch their products become obsolete. Unfortunately, programming GUIs is difficult: "programming complexity tends to increase almost exponentially with user friendliness." [Giguere, 1990, p. 43] "Microsoft Windows and the Macintosh toolbox each have over 400 function calls, about 10 times as many as there are in MS-DOS.... OS/2 Presentation Manager [has] over 1000 function calls." [Urlocker, p. 287] [Thompson, 1989]

> Programming for a GUI with a traditional language is a little like running the Boston Marathon wearing hiking boots and a three-piece suit. Even if you're used to regular jaunts and occasional sprints, be prepared to hit the wall after about 20 miles. In programming terms, that's about 3000 lines of code, or what it might take you to create a half-decent text editor for a GUI." [Urlocker, p. 287]

Availability of Interface Code

In contrast, object-oriented environments have many of these calls already included. Smalltalk's basic Form and BitBlt classes (which are essential to the real-time bit-mapped graphical displays of a DMI) as well as its windows, menus, and scroll bars, are such an integral part of the interface, that the basic building blocks are already in place for other programmers to exploit. It is also tremendously helpful, from a programmer's standpoint, to be able to look at how the Smalltalk developers created, say, a browser, and use the same techniques for other applications. To some, it is no coincidence that the Smalltalk language developed along with the graphical interface, given the complexity of such an interface and the power of object-oriented languages. [Urlocker, 1990] [Wisskirchen, 1986b] [Dodani et. al., 1989] [Ingalls, 1981]

Consistency of the Interface

The consistency provided by pre-existing classes not only affects the programmer, but also extends to the user. If all interfaces are based on the same classes, then menus, windows, scroll bars, and buttons will have predictable behavior from one application to the next. [Wisskirchen and Rome, 1988]

Portability

One advantage of using Smalltalk/V in particular is its portability. GUIs have traditionally inhibited portability because they are so machine-dependent (the Mac Toolbox, for example, is embedded in ROM), and are implemented so differently. There are versions of Smalltalk/V, though, for IBM-compatible 286 and 386 machines, for IBM's Presentation Manager, and for running under Microsoft Windows, as well as for the Macintosh. In each case, only the lowest-possible level of implementation (such as toolbox calls) is altered to allow the interface to conform to its home environment, providing a virtual "machine-independent programming toolkit." [Udell, 1990, p. 228] Smalltalk/V for Windows has menus and scroll bars that look and behave like Microsoft Windows menus and scroll bars; Smalltalk/V for the Macintosh has an interface with the look and feel of any standard Macintosh application. In contrast, Smalltalk-80 running on any machine temporarily turns the machine into an Alto computer clone. [Udell, 1990] [Giguere, 1990]

Smalltalk/V's adaptability means that the consistency of the interface is not restricted to programs written in Smalltalk/V. If the user is accustomed to the Macintosh interface, he won't have to relearn the interface of the Smalltalk/V application. The importance of such consistency cannot be over-stated. A review of drawing applications pointed out one where the interface was so nonintuitive that the reviewer had the sensation of fighting with an

archaic machine in order to create a simple drawing. In contrast, an interface is more likely to feel intuitive if it is based on familiar, well-established interfaces. [Cox and Caldeway, 1987]

Objects Map Into Graphical Objects

The elements of GUIs are good examples of "real world" objects that map readily into object-oriented programming objects. The physical (graphical) notion of a window, for example, is an object with position, size, and several states: open, closed, enlarged, active, or inactive. Each window also has functionality tied to it: a window can move, resize, or scroll. [Urlocker, 1990] [Wisskirchen and Rome, 1988]

Event-Driven Maps Into Message Passing

GUIs are typically event-driven: the user's actions trigger the next action for the application to take. This is in contrast to text-based systems where the programmer imposes his or her own logic on the user, such as: "first choose from menu1, then from menu2," and so on. The general outline of a GUI program is:

```
"program and GUI initialization
loop
    wait for event
    process event
until quit event received
program and GUI cleanup."  [Giguere, 1990, p. 44]
```

> "One of the tricks of successful GUI programming is preparing for the user's unpredictability. A GUI program is by nature passive, always waiting for the user to initiate some kind of action. Several windows may be open at one time.... You simply cannot predict the user's next move, nor should this be attempted. Rather than simply ignoring certain events -- the cardinal sin of GUI programming -- you must engage in defensive programming: disabling menu items, putting up dialog boxes, and so on. And always, always leave the user an escape route." [Giguere, 1990, p. 44]

The passive, event-driven nature of a GUI maps well into the message-passing structure of an object-oriented language. As a dispatcher object

detects a mouse or keyboard action from the user, it can forward that input message to the appropriate object for processing, along with an argument like the mouse location or the key that was pressed. [Cox and Caldeway, 1987] [Urlocker, 1990]

Not a Panacea

Unfortunately, Smalltalk is not a panacea for creating user interfaces. It does have drawbacks, the most notable of which may be the steep learning curve a programmer must climb to become competent in the language. Although its syntax is simple, its collection of classes is not, and in order to become an efficient programmer, one must become familiar with those classes -- what they are, what they do, and more often than not (contrary to the claims of some OOP proponents) *how* they do what they do. Admittedly, the environment helps the programmer conquer these obstacles, but due to the sheer volume of the material, the novice Smalltalk programmer's most valuable resource is time.

White Box vs. Black Box

Ideally, all objects would be black boxes, like the Cat instances above. In order to use an object, one would have to know nothing about it except its interface, i.e., the public messages to which it responds. Then the statement, "There is no need to duplicate or even understand the implementation of the generic behavior in the [superclass]," [Urlocker, p. 288] would be true absolutely.

The real world diverges from the ideal in two ways. First, the interface is not always clear because the names of messages often aren't descriptive enough. I am still not entirely sure what Smalltalk/V's StringModel class intends to do in the message

```
broadcastChangesIn:upTo:withExcess:.
```

Every time I want to know, I have to go back to the code and figure out again what it is doing. (The last time I looked, I convinced myself that it wasn't really broadcasting, and it ultimately ignored the excess.) Part of the problem is certainly documentation; if a comment was not only included in each method (as they generally are) but also clearly explained what the method does (and this was part of the interface) comprehension would be improved. In reality, comments are incomplete, and method names are poorly chosen, unrepresentative of the method's functionality.

The second problem is related to inheritance: it is nearly impossible to create a subclass without intimately understanding the implementation of the superclass. Here the superclass is no longer a black box, because the programmer is doing more than just using its existing functionality; he is changing it. Even the methods that aren't being changed must be understood, because an instance of a subclass will react to its superclass' messages whether or not the programmer understands how those messages work. If the message is somehow inconsistent with the behavior or instance variables of the subclass, run-time errors will result.

MVC and Inheritance vs. Pluggable Views

Smalltalk-80's undocumented but widely used Model-View-Controller (or, in Smalltalk/V terminology, Model-Pane-Dispatcher) paradigm is an example of an area where intimate knowledge of classes is required in order to use them. A window in Smalltalk is made up of one or more *views* or *panes*. At the very least, a window has a top pane, which handles the opening, closing, and labeling of the window, and acts as a middle-manager to all of the window's subpanes. When the top pane is told to show itself, for example, it in turn tells all the subpanes to display. When told to close, it asks all the subpanes if they

are in, or can put themselves in, a state which would enable the window to close (by, for example, saving modifications). [Wisskirchen and Rome, 1988]

The subviews (or subpanes) are, as the name suggests, different ways of looking at an object. One example is a window with two views on a clock: one analog and the other digital. The object being viewed in more than one way is time. The two views are tied together by the object they are viewing, or the *model*. When the number of seconds changes, one would expect both the analog and digital clocks to update. In this way, the two views are obeying a constraint: that both show the same time. [LaLonde and Pugh, 1989]

The *controller*, or *dispatcher* handles the input for each view. There is a one-to-one correspondence between each pane and its dispatcher. If input occurs in the pane, (for example, a new hour is typed into the digital clock) the dispatcher sends the appropriate message to its pane. Panes and dispatchers know of each other's existence, and the pane knows what the model is that it is viewing. Models, on the other hand, only change. The model is set up in such a way that all the views on it are its dependents. When the model changes, it sends out a **self changed** message (or if it changes in a specific way, a **self changed: someWay** message) which is implemented by the Object class. This, in turn, sends out messages to all the dependents (the views) that the model has changed, and the views update themselves accordingly. (The more specific message allows the views to look at the **someWay** argument to see if they should update at all.) [LaLonde and Pugh, 1989]

If the programmer needs to create new pane classes for his application, he will have to fully understand (without the help of even this much documentation) the mechanisms of the MVC framework. That means understanding how all of the relationships and dependencies are set up among

model, panes, and dispatchers; understanding what chain of events will be triggered on a given input; and understanding what information each component of the framework expects every other component to supply. (Central to this understanding is comprehension of how the **self changed** message reverts up to the Object class and back down to the dependents; how panes are added as dependents; and how those dependents in turn get information back from the model, none of which is clear.)

A partial solution is the inclusion in Smalltalk of *pluggable views*. It eventually became clear to Smalltalk's creators that certain views like lists, buttons and text panes were reused often; so as an alternative to creating new panes for every application, views were created that would behave in application-independent ways. List panes, for example, could be lists of anything as long as the model supplied the list. These reusable pane classes, then, have instance variables to hold the application-specific data: data that is "plugged into" the view. [LaLonde and Pugh, 1989] [Dodani et. al., 1989]

While pluggable views are darker boxes, they are still not completely black. The programmer who uses them still needs to have some understanding of the MVC mechanism, and of some of the functionality of the pluggable view. This is true in spite of the fact that the model is only using the panes and is not supposed to be aware of their existence (except as dependents). When the model changes, the panes react to that change by updating themselves. In order to update, however, they must ask the model for some information: what to display. A list pane, for example, will ask the model for the list of things to show. The programmer must know that the list pane is going to react this way, and provide a method in the model to supply the list. The pane's reaction is not part of its interface (its list of messages); rather, it is hidden in the code of the

methods. The programmer must look through and understand this code to find out how to properly use the pane.

(Understanding is another issue altogether. When the pane updates, it sends its model the obscure-looking message **model perform: name**. **Perform:** is a method in the Object class, which takes the argument (in this case, whatever **name** is), and treats it like a message to be sent to the model. In this round-about way, the model can have more than one list pane that, with different names, can ask for different lists to show.) Fortunately, Digitalk has realized these deficiencies in MPD, and has developed a new paradigm based on an ApplicationWindow class for its new Smalltalk/V for Windows and Smalltalk/V PM releases. (For a detailed description of this paradigm, see [LaLonde and Pugh, 1991].)

Object-Oriented vs. Classical Standards

In spite of Smalltalk's steep learning curve in areas relevant to the user interface, it is still easier to program than traditional languages when building GUIs. In more general graphical applications, however, (even when user interfaces aren't the main focus) object-oriented languages still show their superiority; in this case, that superiority is over classical graphics standards.

GKS and PHIGS Origins

Classical graphics standards such as the Graphical Kernel System (GKS) and the Programmer's Hierarchical Interactive Graphics Standard (PHIGS) grew out of conventional, procedural languages like FORTRAN. This foundation necessarily limited the standards to accommodating procedures and functions, but not objects. Not only is it difficult (but not impossible) to make objects conform to the standards, but objects themselves present such an elegant solution to the management of graphical data and functionality, that such a

conformity feels like a step backwards. [Wisskirchen and Rome, 1988] [Hopgood et al, 1983] [Shuey et. al.]

Naming Problem

Because objects can store their own data, the management of objects (in this case, graphical objects) in an object-oriented language is cleaner than in the standards. One example is in the naming of objects. In GKS, the closest one gets to objects is the segments. A segment can be created, opened, filled with the commands which draw it, then closed. The segment is only a list of procedures (not data), so as the list of segments is managed on a global scale, so must the name of each segment be managed globally. Naming conflicts must be explicitly avoided because each name is, by necessity, globally available. [Wisskirchen and Rome, 1988] [Hopgood et. al., 1983]

If, for example, a picture of a village is made up of House1, House2, and House3, and each house has its own doors and windows, the doors and windows must also be named Door1, Door2, Door3, Window1, Window2, etc. A better solution is offered by House objects that hold, internally, their own door objects. Here, each door can be named, simply, "door." Since the door of one house is not visible to the door of any other house, no naming conflicts can occur. Additionally, the house can hold its own name, so the village does not need to painstakingly associate each house's name with the code that draws it. Rather, it can hold just a list of houses, and ask any house for its name when the need arises. Furthermore, because an object's data is persistent, the code that draws a graphic needs to be executed only once, not every time the object is referenced. [Wisskirchen and Rome, 1988] [Wisskirchen, 1986a]

Localized Data

In general, any time it becomes necessary to manage all the elements of a graphics system on a global scale, the management will be more difficult than

if some of the organization could be delegated to objects. In a drawing application like that of this thesis, where grouping of objects into larger, composite objects is permitted; graphics, and the structures that hold them, can become arbitrarily complex. If the implementation is an object-oriented one, though, a CompositeObject class can be created to store the pieces of the composite. It is as if the pieces, however complex, are primitives like any other. In contrast, "for generating a [MacDraw-like grouping] system with GKS, the entire process would have to be programmed...and the application programmer would have to bridge the gap between that complex segment structure and the linear structure of the GKS segments." [Wisskirchen and Rome, 1988, p. 31] [Fuchsberger and Krasemann, 1990]

Constraints

Local storage of data also means constraints can more easily be followed. Just as their are often constraints among subpanes in an interface, composite objects must also obey constraints. Composite objects can be moved or stretched: when moved, all the objects that make up the composite must maintain their relative positions; when stretched, their relative sizes. When the data is local to each object, the objects can be asked to update their positions or sizes each in the same way. "In classical systems, it is very complicated to formulate such [constraint] relations, and the solution must be programmed procedurally." [Wisskirchen and Rome, 1988, p. 77]

Application-Specific Data

PHIGS, developed after GKS, has some advantages over the earlier standard. For example, it does allow the reopening of segments for modification, and, as the name implies, it allows for a hierarchical arrangement of graphical elements. A bicycle drawing may include calls to wheel drawings, for example. PHIGS also attempted to provide a facility for including application-

specific data with the segments; for example, tire pressure could be associated with each wheel. [Wisskirchen and Rome, 1988] maintain, however, that it is cumbersome to store data within the drawing hierarchy and still worse to maintain two hierarchies (one with drawing calls and one with data) and build a bridge between them. They consider it a far cleaner approach to store the application-specific data within an (object-oriented) object which, by definition, holds both data and functionality. [Wisskirchen and Rome, 1988] [Wisskirchen, 1986a] [Wisskirchen, 1986b] [Shuey et. al.]

Animation

A large application for the storage of data with an object is in the area of animation. Any animate object can store its current position and orientation. Further, animate objects can be linked with specific behaviors which allow them to move independently of other objects. They would not have to be controlled globally; rather, each could contain its own impetus for movement. One possible extension of this thesis, for example, is to build an interface which allows the user to interactively create an animation. As the user drags or stretches an object, the object records the movement and speed of movement, which can all be played back at a later time.

Has-a vs. Is-a Hierarchy

The distinction should be noted between the PHIGS hierarchy and an object-oriented class hierarchy. The PHIGS hierarchy is a "has-a" or "part-of" hierarchy; whereas the object-oriented inheritance hierarchy is an "is-a" hierarchy. In the bicycle example above, the bicycle *has* wheels as parts; each wheel is *part of* the bicycle. A subclass, though, *is* whatever its superclass is, but more specialized. A Kitten *is a* Cat (is an Animal, is an Object). The distinction is important from an implementation point-of-view. If, for example, the programmer knows how to draw an ellipse, and a circle *is*

an ellipse with both focal points the same, most of the code exists for drawing the circle, and the specialization is simple. On the other hand, if a drawing application supplies the grouping function mentioned earlier, a complex drawing can be composed of several component parts: a wheel can be made up of lines and circles, a bicycle of wheels and frame. Any composite object will be made up of simpler objects by, perhaps, having as an instance variable a list of those parts. While the part-of hierarchy is a feature of the application important to the user and must be managed, it provides no help in building the application.

MacDraw Observation

A final note concerning the limitations of drawing programs created in conventional languages involves an observation I myself made while using the MacDraw program. MacDraw is the original object-oriented drawing package for the Macintosh and, by using it, one can glean some information about its implementation: most notably, that that implementation is not an object-oriented one. As was stated earlier, objects in object-oriented drawing programs are defined by mathematical expressions. A line, then, would be stored as its two endpoints since, mathematically, this is all that is necessary to recreate the line. MacDraw offers two types of lines: those that can be drawn at any angle, and "orthogonal" lines: lines that must be either horizontal or vertical. If the user creates an any-angle line, then picks up one end of the line to stretch it or adjust the angle, he may (intentionally or inadvertently) release that end point at the same x- or y- coordinate as the stationary end point. Unfortunately, the line is now orthogonal! Try as he might, the user cannot pick up either endpoint again and stretch the line to any angle. It will always remain either horizontal or vertical. (In fact, in the attempt he may lay one end point on top of the other, and wind up with an unstretchable

point!) It is obvious from this that MacDraw *only* stores lines by their endpoints, and once either the x- or y- values of those endpoints are the same, the line is, by definition, orthogonal. Had the lines been instances of the classes Line and OrthogonalLine, however, they would have existed in a framework which allowed them to remember their own constraints. Orthogonal lines would always be orthogonal, and any-angle lines would always be able to take on any angle even though they may temporarily look either horizontal or vertical.

Related Work

Object-Oriented Concepts used for GUI's

Windows

The Smalltalk environment is not the only graphical user interface to be built in an object-oriented language. Other GUI programmers are realizing the benefits of applying object-oriented techniques to the complex task of creating these interfaces. Microsoft Windows Software Development Kit, for example, provides the application programmer with an object-oriented approach to creating windows. When a window is created, "its window class must be defined and registered." [Giguere, 1990, p. 51] All the windows of the application are then subclasses of that class. As events are generated, Microsoft Windows treats the events like messages, sending them to the appropriate window. [Giguere, 1990] [Urlocker, 1990]

MacApp

MacApp is an object-oriented framework for creating applications on the Macintosh. Like Smalltalk's MVC paradigm, MacApp's is a white-box framework, which provides application-independent windowing behavior, and allows the programmer to "plug in application-specific details, such as the

contents of each window." [Dodani et. al., 1989, p. 258] MacApp's class hierarchy includes TView for describing the behaviors of all display objects, and TWindow, a subclass of TView, for defining standard windowing behavior. A notable application built successfully with MacApp is Brown University's Intermedia (a hypermedia system). Its designers praised the object-oriented design of MacApp for speed in development, making code sharing possible, and enforcing consistency in the code developed by independent programmers. [Yankelovich et. al., 1988].

NextStep

NextStep is an object-oriented development environment created to "simplify the design and creation of the event-driven interface" [Thompson, 1989, p. 265] of the NeXT computer. Its Application Kit provides 38 predefined objects, some of which may be used unaltered, while others must be adapted to the application. NextStep also includes a truly black-box user interface construction set called Interface Builder (IB). IB allows the developer to build his application's interface with "a series of mouse-clicks and keystrokes," [Thompson, 1989, p. 266] including building the bridges between the objects of the interface and the application code. The Interface Builder's interface is much like an object-oriented drawing package, where buttons can be picked and stretched, for example, or the text in a button can be selected and changed. The links from interface objects to application objects are also represented visually as they are created. [Thompson, 1989]

User-Interface Construction Sets

Interface Builder is not entirely unique. [Dodani et. al., 1989] define *user interface construction sets* as "collections of high-level tools that let you interactively configure and construct user interfaces...[usually] built upon [white-box] frameworks" like Smalltalk's MVC framework. [Dodani et. al., 1989,

p. 256] ViewEdit is such a construction set for use with MacApp. With MacDraw-like tools, rectangular areas of the interface can be created, selected, and moved.

Glazier

Glazier is a construction set for "interactively building windows for Smalltalk's standard pluggable views." [Dodani et. al., 1989, p. 262] Like ViewEdit, Glazier allows the programmer to create and relocate subviews; like Interface Builder, it lets him interactively form the dependencies between the model and its panes. Furthermore, Glazier automatically generates subclasses to correspond to the newly created windows. [Dodani et. al., 1989]

IconMaker

IconMaker is an older application with the functionality of Glazier (where all the elements of the user interface are referred to as icons). Like Glazier, IconMaker provides tools for both creating the interface and defining the dependencies. "Dependencies between icons are specified in two steps. First the icon and its dependent have to be selected.... Then a property sheet will appear that asks for the [appropriate] message pattern. A warning will appear if the message is not in the set of messages the dependend [sic] icon understands." [Kramer,1984, p. 197] The creators of IconMaker had visions of extending it to programming-by-example, where it might be possible to "examine the user actions, extract routine work, and offer the user some shortcuts in performing her task." [Kramer,1984, p. 198]

Object-Oriented Concepts used for Graphics Applications

Animation

Some of the very first Smalltalk applications were in graphics, and the relative ease of producing animation in Smalltalk has been a draw ever since. [Deutsch, 1989] A useful application of Smalltalk's graphical capabilities, from

an educational point of view, is using it to animate programs and algorithms. [Mohamed, 1987] and [London and Duisberg, 1985] both used Smalltalk to animate algorithms, the former for illustrating array sorting, the latter for a number of algorithms ranging from sorting to circular queuing (the "producer-consumer-ring buffer system" described in that paper).

Another animation example is the Actor/Scriptor Animation System (ASAS), which was used to create graphical sequences for the Disney movie *TRON*. Implemented in Lisp using object-oriented concepts, ASAS uses the idea of actor-objects, each with independent movements and motivations (scripts). "ASAS actors are participants in the animation, communicating by sending and receiving messages. Once an actor is started, it remains a part of the animation until it stops itself or is stopped by another actor." [Lorensen et. al., 1987, p. 5]

The *O*bject-oriented *SC*ene *A*nimato*R*, (or OSCAR) also relied heavily on object-oriented design, although it is implemented in C. Used for 3-D simulations, OSCAR makes use of "an object-oriented script language as a user interface" for creating, controlling and managing the animations. [Lorensen et. al., 1987, p. 7] This reference is, in fact, part of an entire volume of SIGGRAPH course notes devoted to using object-oriented techniques for geometric modeling and rendering. While none of the implementations therein employ object-oriented interfaces, and most are written in C, they are testament to the benefits of using object-oriented design for graphical applications.

Smalltalk with PostScript

Smalltalk has also been successfully combined with the powerful printer command language, Postscript, to create what [Haaland and Thomas, 1989] termed "SmallScript." The goal of that project was to create a user

programmable framework similar to HyperCard, but with Smalltalk's interactive debuggers, and object-oriented graphics instead of bit-mapped. To do this, the authors implemented an new class, PostScriptPen, to "encapsulate the interface between Smalltalk and PostScript. Messages are sent to PostScriptPen when graphical output is desired." [Haaland and Thomas, 1989, p. 58] In SmallScript, this output is sent to either printers or the terminal display.

Drawing Packages in Object-Oriented Languages

As for object-oriented drawing packages written in object-oriented languages, the examples are few. The earliest was perhaps Ivan Sutherland's 1963 *Sketchpad*. Without any official object-oriented language at that time, Sketchpad uses object-oriented concepts. Instances of graphical objects were interactively created, for example, by applying geometric transformations to class (called "master") definitions. [Lorensen et. al., 1987]

ObjectDrawing -- Reusing the FreeDrawing Class

A more recent example which, at first glance, appears to be very much like this thesis is that of [Fuchsberger and Krasemann, 1990]. Entitled "ObjectDrawing -- Reusing the FreeDrawing Class," their implementation even seems, at first, to be like mine. My primary class also inherits from the FreeDrawing class, a bit-mapped drawing class provided with Smalltalk/V's tutorial. (Once instantiated, the class allows the user to create lines, circles, ellipses and free-hand drawings; change the pen thickness; copy rectangular areas of the bitmap to the clipboard; paste bitmaps from the clipboard; erase parts of the bitmap; and zoom in on the bitmap to edit individual bits using another class, BitEditor.) My implementation approach and [Fuchsberger and Krasemann]'s were the same: "to construct a MacDraw-like tool from a MacPaint-like class;" and I was motivated, as they were, by the need to

manipulate graphical objects, not just bits. [Fuchsberger and Krasemann, 1990, p. 10] We also use a similar collection of classes. Besides the *ObjectDrawing* class, [Fuchsberger and Krasemann] also created a *DrawObject* class "with its subclasses *RectangleObject*, *EllipseObject*, *LineObject*, *IconObject*, *TextObject*, and *CompositeObject*." [Fuchsberger and Krasemann, 1990, p. 10] CompositeObjects are objects created from collections of other objects, grouped together to be treated as one.

Perhaps because it is implemented in a different version of Smalltalk/V, the FreeDrawing class of [Fuchsberger and Krasemann] differs somewhat from the FreeDrawing class which was available to me. First, theirs is a subclass of the Pen class, whereas mine is a subclass of Object. It is unclear from the paper how much difference this makes in the implementation; the authors make the statement that "*FreeDrawing* inherits [much] of its functionality from the class *Pen*," without stating what that functionality is. [Fuchsberger and Krasemann, 1990, p. 12] Their FreeDrawing does include the abilities to change text fonts and input text, change the pen color, and draw rectangles, all of which were absent from my FreeDrawing class. Perhaps the pen color, pen size, and BitBlt transfer modes are inherited.

[Fuchsberger and Krasemann] more faithfully adhered to the restrictions of the FreeDrawing class than I did. They modified the implementation (but not the behavior) of FreeDrawing somewhat to separate its input functionality from its output functionality. Then they created the ObjectDrawing class for the management of graphical objects. Since FreeDrawing's input and output were separated, ObjectDrawing could inherit both, and add some additional input features (selecting, deleting, etc.). Between input and output, ObjectDrawing would store the newly created object in its object list. Finally, they created the abstract class DrawObject to define common graphical object

behaviors, then specialized that for the individual types of objects, including CompositeObjects.

My thesis also includes an abstract class analogous to DrawObject, and similar subclasses. Unlike [Fuchsberger and Krasemann], however, I did not modify the FreeDrawing class. Consequently, although my ObjectOrientedDrawing class is technically a subclass of FreeDrawing, the latter's functionality was so limited that I ended up using it more as a launching point than for code reuse. Its presence was vital for implementation examples, but even the input methods were ultimately overridden to allow for the proper storage of graphical objects.

One of the problems [Fuchsberger and Krasemann] encountered was that of scrolling the graphics window. "Scrolling as it is implemented in the FormEditor makes the positions of the DrawObjects images within the ObjectDrawing window inconsistent with the positions of the DrawObjects themselves." [Fuchsberger and Krasemann, 1990, p. 16] To avoid this problem, [Fuchsberger and Krasemann] chose to restrict scrolling by creating a dispatcher without scrolling ability. I did not make the same choice; I have implemented scrolling, updating the graphical objects' positions.

Flavors Graphics Editor

Closer to the intended functionality of my thesis is that of the graphics editor of Flavors, the object-oriented Lisp extension whose first application was "the construction of the windowing system of the Lisp machines." [Wisskirchen and Rome, 1988, p. 103] Flavors includes as part of the language both a graphics windowing system (*gwin*) for displaying graphics (including those that are part of the interface) and a graphics editor (*ged*) for creating graphics. This graphics editor is object-oriented. [Texas Instruments, 1987a] [Texas Instruments, 1987b]

The graphics windowing system includes methods (flavors) of the class Windows to provide standard windowing behavior: the bitblt combination rules used, for example, or the colors of the foreground and background. A mixin (a *mixin* is a class resulting from multiple inheritance) of both windows and graphics has methods for drawing graphical images on windows -- images ranging from an entire picture list, to individual points, lines, circles, polygons, etc. [Texas Instruments, 1987b]

The graphics window package, GWIN, "includes instantiable flavors to create windows or panes that allow graphic objects." [Texas Instruments, 1987b, sect. 12, p. 34] Examples of these flavors include the ability to insert graphical objects into the list, find the distance from a particular graphical object, find the point nearest an object, learn whether lines intersect, and so on. Graphical objects in general can draw themselves on windows, move, scale, and undraw, and report their distances from given points. Specific objects, like rectangles and arcs, have specialized drawing and displaying flavors. [Texas Instruments, 1987b]

The Flavors graphics windowing system is a complex system with a very rich, complete set of classes and methods. The Flavors graphics editor is also powerful as a drawing tool. It includes the abilities to create arcs, sectors, cubic splines, triangles, and rulers in addition to the standard drawing package features of circles, rectangles, polygons, free-hand drawings, and text. The editor also seems to have some "awareness" of the underlying pixels. Zooming in and out is allowed, for example, and line widths can be made to scale with the object or maintain their original thicknesses. Both arrow keys and the mouse can be used for cursor movement: an arrow key will move the cursor one "unit", where a unit is defined as a pixel if the grid is off, or a grid point if the grid is on. Another nice feature is the ability to undo any of a list

of previously committed actions, in the opposite order in which they were done; most drawing packages allow the user to undo only the last operation. [Texas Instruments, 1987a]

The graphics editor allows grouping into subpictures, gray scale fills, and the specification of bitblt combination rules for combining an object with its background. It also offers the ability to load in a previously-created picture as a background picture to be used as a template or for comparison against the foreground. A feature it shares with this thesis is the concept of a graphics presentation, where several drawings (pages) can be grouped together into a logical unit. (Their user interface for this functionality differs greatly from mine, however.) [Texas Instruments, 1987a]

Three possible limitations of the Flavors graphics editor are that: 1) it allows no drag-scaling, where an object stretches like a rubberband with the movement of the mouse (instead, objects are scaled to some fraction or multiple of the original size through dialog boxes); 2) it does not allow the readjustment of vertices of polygons, curves and polylines; and 3) there is no rotation of objects, either in 1° or 90° increments. Even clearer drawbacks of the package, though, are larger-scale. First, it exists only as part of the Lisp/Flavors language available only on Lisp machines. It is not portable. Second (and perhaps a result of the first), it is designed for Lisp programmers. It was never intended as a general-purpose drawing package for naive users. Its documentation reflects this fact: the editor must be loaded by evaluating a Lisp expression, and the documentation addresses issues of buffer storage and buffer-list editing. [Texas Instruments, 1987a]

Drawing Packages, Features and Criticisms

Features

The idea of a general-purpose, interactive drawing or painting package with a choice of drawing tools was probably first dreamed up at Xerox PARC. Before their ToolBox, with its rudimentary COPY, DRAW, ARC, BLOCK, and LINE tools, graphics were generally computer generated through programming. Macintosh drawing and painting packages were the first to be commercially successful. MacPaint was the first bit-mapped graphics application, introducing palettes and icons; MacDraw the first object-oriented package. Since then, many software companies have followed suit; today there are quite a number of drawing and painting packages available, with a large variety of features. See the tables in Appendix A for a comprehensive list of these features. [Bowman and Flegal, 1981] [Robinson, 1987] [Cox and Caldeway, 1987] [Guzelimian and Mello, 1986]

Complaints

Criticisms of existing packages fall into two categories: a lack of features, and a poor user interface. Strong text manipulation facilities, for example, are important to graphic designers. Packages without kerning, leading, adjustable letter spacing, multiple type fonts and faces per block, or the ability to rotate text are therefore not recommended for these users. Likewise, curve editing is important to some users. If curves cannot be created, combined, reshaped or scaled, that package will draw complaints. [Guzelimian and Mello, 1986] [Burns and Venit,1987] [Kerlow, 1989] [Kleinman, 1989] [McKinstry, 1989] [Robinson, 1987] [Roth, 1989]

While effective drawings can be created in packages without all of the possible features, poor user interfaces are inhibitive. Some of the criticisms leveled against interfaces involve the relative difficulty of creating objects.

Adobe Illustrator, for example, implements all freehand drawing as Bezier curves. This is because Illustrator is tied to PostScript, the printer command language also created at Adobe, and Bezier curves are definable in PostScript, whereas arbitrary freehand drawings are not. Unfortunately, it is difficult to be precise with Illustrator's freehand tool. One reviewer stated that it was "next to impossible...to draw curves without working through Illustrator's tutorials first." [Kleinman, 1989] Another labeled the freehand curves as "jumpy", "unintuitive" and "hard to control," requiring 20-30 hours of practice. [Kerlow, 1989] Paths that are hard to join, handles that are difficult to grab, and transformations that apply only to individual objects (and not groups) are also criticized. [Guzelimian and Mello, 1986] [Burns and Venit,1987] [McKinstry, 1989] [Roth, 1989]

Other interface-related complaints are of a lack of an Undo feature, or one that undoes deletions only; text or drawings that are not WYSIWYG; no keyboard shortcuts for often-used operations; commands that are available *only* through the keyboard, with no menu equivalents; a small or crowded work area; unclear distinction of layers; and operations that involve too many steps. Concise menus are vital. One package makes heavy use of submenus and sub-submenus requiring "great manual dexterity (not to mention total recall) to use effectively, [creating a] bewildering user interface employing a blizzard of menus (pull-down, pop-up, concatenating -- you name it, Scoop has it, and frequently simultaneously)." [Cline and Stern, 1988, pp. 14-15] Other interfaces are simply labeled "unintuitive" by reviewers. SpaceEdit's scroll bars seem to work backwards, and its line drawing requires clicks at both ends of the line, instead of the usual click-drag-release mechanism. [Robinson, 1987] LaserPaint's interface is a whimsical machine with levers, buttons and thumb wheels which ends up being inconsistent and "hinders creativity and

productivity." [Cox and Caldeway, 1987, p. 105] [McKinstry, 1989] [Burns and Venit,1987]

Results

The goals of this thesis were twofold. On one hand, I wanted to exploit the advantages of an object oriented language for the creation and management of graphical objects, including complex graphical objects like composites. Secondly, I wanted to present a user interface which combines the advantages of both object-oriented and bit-mapped graphics. To restate those advantages, that of the object-oriented interface is the ability to manipulate objects simply by selecting them with a mouse click, then stretching them, moving them, etc. The advantage of a bit-mapped interface is the ability to manipulate individual bits. Bits can be erased and added, usually through a zoomed-in view of the drawing: a bit editor.

Bit-Mapped/Object-Oriented Integration

As stated in the introduction, very few packages have attempted this combination. One which has been somewhat successful in its attempt is SuperPaint. SuperPaint offers two layers: a paint layer and a draw layer (patterned closely after MacPaint and MacDraw, respectively), each with its own separate tool palette and behaviors. Bitmaps can be cut from the paint layer and pasted into the draw layer (where they are still bitmaps), or cut from the draw layer and pasted into the paint layer (where they become bitmaps). The separation of the two layers has been criticized: since they are so weakly integrated, people will tend to use the layers independently, so that the application offers little advantage over having two separate draw and paint packages. [Cox and Caldeway, 1987] SuperPaint 2.0 has attempted to make the integration stronger by including an auto-trace feature. Bitmaps on the paint

layer can be automatically traced into the draw layer, where they become objects which can be scaled and manipulated. SuperPaint has a difficult-to-control curve tool, though, and no provision for creating a white bit-mapped object of an arbitrary shape (other than rectangular) for pasting into the draw layer. This combination makes it difficult to paste a white curve precisely over a portion of a black curve to, in effect, "erase" that portion. [Fenton, 1989b] [McKinstry, 1989]

With my package, the user can also create bit-mapped objects out of object-oriented ones, but without going to another layer, and the new bit-mapped object is not restricted to only one (or even an entire) object. Any arbitrary rectangle can be copied from the drawing surface (using the selection box) and pasted back in as an object, where it can be selected, stretched, grouped with other objects, or moved. (See Appendix B.)

If an object is one of these "paste objects", portions of it may also be erased. With a mouse click, the eraser tool will white out portions of the object in an area the size and shape of the eraser cursor. (See Appendix B.)

Bit Editing

I wanted to make it possible for the user to bit-edit any object, however, not just paste objects. One solution would have been to force the user to make any object he wanted edited into a paste object first, by copying it out of the bitmap. Had this been the case, however, the object would not be able to revert back to its prior existence as a mathematically-defined ellipse, for example.* From that point on, it would be a paste object, and I would have provided little functionality beyond that of SuperPaint.

---

* The closest I might have been able to come would have been a series of Bezier curves, had I chosen to implement an auto-trace feature.

The approach I chose instead was to provide an interface which allows the user to view the drawing at a higher level of magnification in a bit editor, where point objects can be applied near or on top of the object being edited, and may then be grouped with that object. I needed to consider what kind of object would result from such an operation, and what transformations would then be available on such an object. Is the resulting object fundamentally the same object as the original, but with embellishments? Can it be reverted back to its original shape? Does it conform, as the original did, to some mathematical definition? These were all questions I needed to address. I also needed to explore how the answers to these questions would be be made obvious to the user. I feel that my application not only answers these questions, but does it in a way that is consistent both from a theoretical standpoint, and in its user interface.

Point objects

By adding to the object-oriented model a single concept, I was able to link together the advantages of the bit-mapped and object-oriented interfaces and, I found, still maintain the mathematical model of the traditional object-oriented package. The concept is that of the point: a basic geometric element. Other object-oriented packages ignore the single point as graphically significant. No provisions are made for creating, removing, or manipulating points as objects.

Freeform objects

As it turned out, the addition of point objects provides a convenient way of fitting another type of object into the mathematical model. Although all object-oriented graphics packages include some kind of drawing tool for creating "free-form" objects (it would be nearly impossible to accommodate all drawing needs without one), they grapple with different mathematical

justifications for the resulting object. In some packages the result is a small bitmap, with no geometric equivalent. In others, it is either a series of line segments or a series of Bezier curves. (Some packages even insist that the user create such objects with a Bezier curve tool, a tedious process at best.)

If the point object is included in the mathematical model, however, a bitmap *does* have a geometric equivalent: it is a collection (a composite) of contiguous points. Just as any other objects (lines, circles, etc) can be grouped together to form a composite object, points are (conceptually) grouped together when the user creates a free-form object.

Point Creation and Manipulation -- the Bit Editor

The inclusion of the point object introduces a number of new considerations, relating primarily to the user interface. Fundamentally, since point objects are now recognized as individual objects, provisions must be made for creating, manipulating and removing them, as individuals. This is the role of the bit-editor.

Role of Points

I knew I wanted point objects long before I realized their significance in freeform objects. Since point objects are so tiny, I rationalized that it only made sense to create them through a bit editor. Also at that time, I considered a point's only function to be the embellishment of pre-existing objects. That is, points would be there to enable the user to do finely detailed editing of circles, lines, etc., by placing white and black dots on top of or near these other objects.

First Interface attempt

My first implementation of the bit editor allowed the user to select an object and choose **BitEdit** from the **Tools** menu. A bit editor opened showing that object magnified eight times, and presented a palette of tools for adding points,

lines and other embellishments to the object. When the user was finished editing the object, he could leave the bit editor by choosing to either **Cancel** the bit editing operation, in which case all of the new point and line objects would be discarded; or by choosing to **Group** the new objects with the original object.

User Interface Complications

Because of the limitations of the windowing surface, not all objects could be shown in their entirety at eight times their original magnification. For this reason, I needed to allow the user to scroll the bit-editing window in either direction, up to the size of the original drawing. Unfortunately, this introduced a potential problem. If the user invoked the bit editor on one object, then scrolled to a far distant object and began adding points and lines; upon leaving, the points and lines would get grouped with the original object, not the one they were created near. The problem, as it turned out, was not a result of allowing scrolling, but of forcing the bit editor's closing to automatically group all new objects with the original object. Instead, the user should be allowed to choose which objects are to be grouped together, just as he can in normal, zoomed-out mode. (This also allows hierarchical groupings within the bit editor. With automatic grouping, the user had to go out of the editor and back in again to have composites of composites.)

Second Interface

My second implementation allowed the user to select an object, choose to bit edit that object and, in the bit editor, add points and lines on top of and around the object. He could group those points and lines with the object if he chose. Then, while remaining in the bit editor, he could scroll to other objects to add points and lines to those.

Point Creation within Freeform objects

When I discovered that freeform objects could be thought of as collections of point objects, I realized that points are created in yet another manner, outside of the bit editor. When the user chooses the drawing tool, he creates a single freeform object whenever he presses the mouse button, moves the mouse, and releases the mouse button. (This is the traditional freeform creation interface: one mouse-down/mouse-move/mouse-up sequence constitutes one freeform object.) The freeform object is, then (conceptually, if not in implementation -- see below), a collection of point objects which are automatically grouped together.

Freeforms vs. Composites

Actually, freeform objects can only loosely be considered composite objects. Unlike composites, they are monolithic, containing only point objects, whereas a composite could be made up of a number of different types of objects, even smaller composites. Also, freeform objects are the only objects which are created already in a grouped state. Other composites are explicitly grouped via a menu command. I further widened the gap when I decided that, unlike composites, freeform objects could not be ungrouped into their points in the normal, zoomed-out mode. For that, the user must go to the bit editor. The bit editor is the only place where point objects can exist in an independent, malleable state.

Freeform implementation

Freeform objects are actually implemented as bitmaps. It would pose entirely too much overhead to store every freeform object as a collection of point objects. With the restriction, however, that points are only manipulable in the bit editor, the transition from bit-maps to points has a clear boundary for both the implementation and the user. It is when the user chooses to bit-

edit a freeform object, and even then only when he ungroups the freeform object within the bit editor, that the points of the bitmap become actual point objects. (I considered making freeform objects automatically ungroup into their points when they entered the bit editor, but rejected this idea for the same reasons I rejected the automatic grouping of objects when the bit editor is closed. The user should have as much control as possible over which objects are grouped together and when.)

Points vs. Edges

In order to make clear to the user that this decomposition of a freeform object has occurred, it was necessary to find a representation of the independent, ungrouped point objects that distinguished them from the "fat-bits" that make up both grouped freeform objects and the edges of other objects (circles, lines, etc.) in their expanded state. To do this, I made the point objects slightly smaller: six pixels square instead of eight, which added a one-point-wide gap around each edge. This is enough to see both that they are different kinds of points, and that they are separate from each other. A freeform object and several point objects are shown in figure 7.

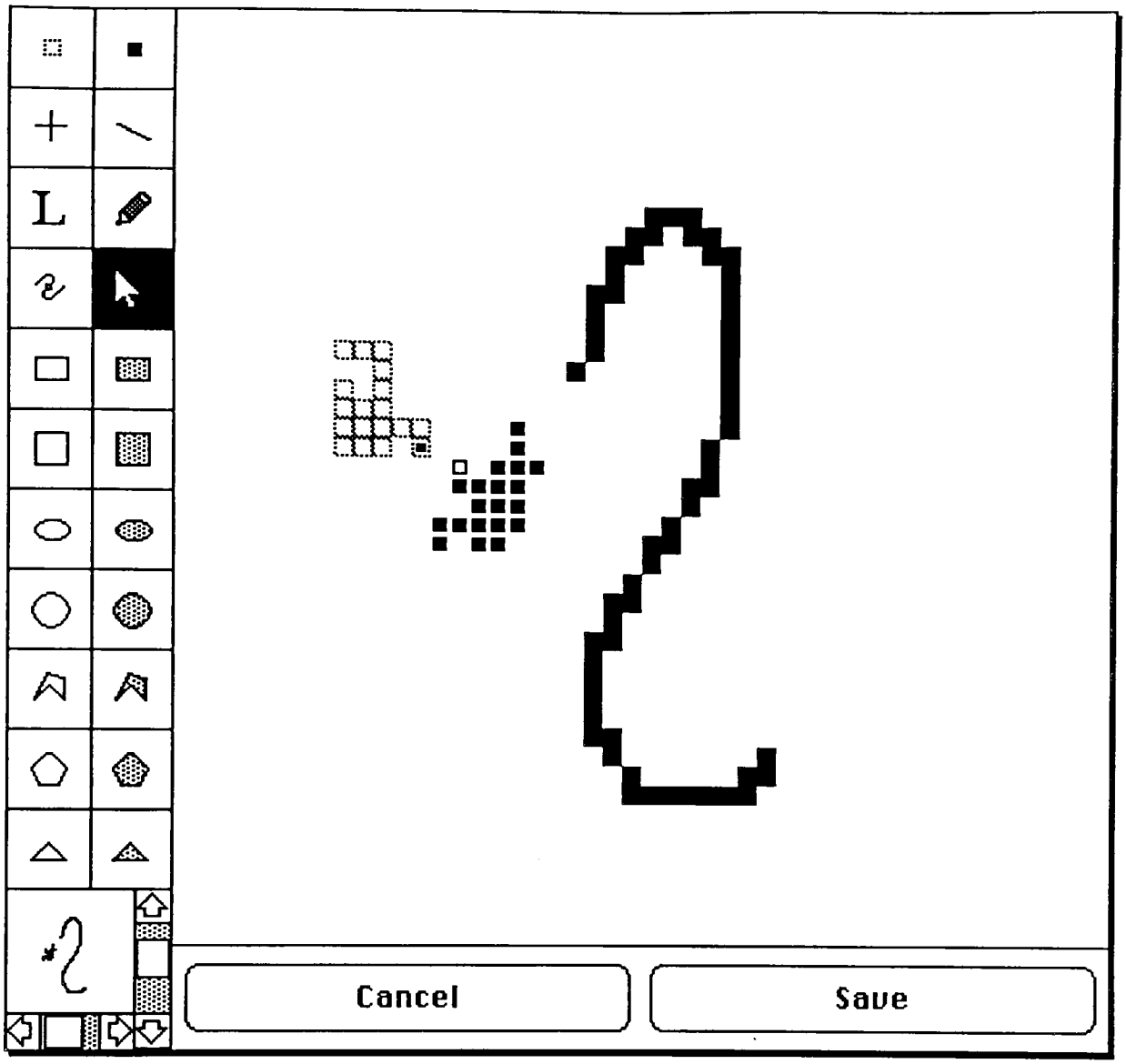


Figure 7 -- Bit Editor with Freeform and Point objects

Current Interface

Now the user can select any object, choose to bit edit that object, and scroll to any other object while still in the bit editor. He may at any time choose to ungroup a composite object and manipulate its components separately (just as in the zoomed-out mode). In the bit editor, though, he may also ungroup a selected freeform object. In this case, it will get reduced to its component point objects, which will change size to indicate that they are now separate

and independent. Of the basic objects, only freeform objects can be ungrouped into points. Lines, circles, rectangles, etc. are already at their lowest level of decomposition. This is made clear by the **Ungroup** selection on the **Arrange** menu: it is enabled when composite and freeform objects are selected in the bit editor, and disabled otherwise.

Point Creation

Point objects in the bit editor can be manipulated the way any other objects can be manipulated in the zoomed-out mode, except that they cannot be stretched. New points can be created with the point tool in a manner very similar to the creation of freeform objects. The user may create one point at a time by clicking the mouse button without moving it, or create several by holding the mouse button down while moving it. The difference between this tool and the freeform tool is that, with the point tool, the points are separate (smaller), not grouped.

Point Manipulation

Existing points may also be selected in the bit editor with the selection tool. Since handles indicate both selection and the direction(s) in which objects can be stretched, point objects have a single handle -- in the middle -- to show that they are selected but not stretchable. (For an explanation of my decision to disallow point stretching, see below.) On white points, the handle will appear black, and on black points the handle will appear white. One of each color is selected in figure 7. Once selected, points may be picked up and moved, or deleted. If more than one point is selected, they can be grouped, at which time they become, once again, components of a freeform object. (From an implementation standpoint, they revert back to bits in a bitmap.) This change is represented by them becoming larger and thus (if they are contiguous) connected. Point objects can be either black or white; white points laid atop

the bits that make up, say, the edge of an ellipse will make the ellipse appear to have part of its edge removed. (The edge is actually still there, but obscured.)

Point Grouping

The user is not restricted to selecting only points for grouping. He may select several points, a line, and a circle, for example, then choose to group everything in the selection. If there is more than one point object in the group, however, the points will be grouped together into a freeform object. In the above example, then, the user would have a group consisting of a line, a circle, and one freeform object; if he ungrouped this composite in the zoomed-out mode, these are the three pieces he would be able to move independently. Once again, points may not exist independently in the zoomed-out mode, in order to reduce overall memory requirements.

Segregation

In fact, since grouped points are implemented as bitmaps, the points themselves may have no independent attributes once they are grouped. This means that black points and white points cannot be grouped together. The resulting freeform object may be either black or white, but in order to avoid storing the color information for each point, the point objects in the group must be the same. If the user attempts to group together points of different colors, a dialog box will appear asking him if he wants to make all the points in the grouping black, all white, or cancel the grouping operation. If he cancels, he may arrange a different grouping.

Automatic Grouping

Since points may not exist independently outside of the bit editor, any points left ungrouped when the user attempts to leave the bit editor will be collected into as many freeform objects as there are colors of points (one or two). Even if there are only black points, this restriction poses a few user-

interface complications. First, the user might ungroup a freeform object, group back together only half of its points, then leave the bit editor. The other half of the points will automatically be grouped into a second freeform object. In another scenario, the user may ungroup a freeform object, add several points, and leave the bit editor. The original object and the new points are now one single object The original points are not distinguishable from the added points, since there is no hierarchy to the grouping. Finally, the user may add points to one area of the drawing, scroll to a far distant area, and add more points. If he has done no grouping of his own before he leaves the bit editor, there will be a large freeform object which has a few points in two clumps quite a distance from each other.

At first glance, it may appear that I have recreated the same problem I tried to resolve by removing the feature which would automatically group an object with its bit-edited points upon the closing of the bit editor. There are two differences, though. First, unless the user has ungrouped a freeform object, the new points are not grouped with the original object selected for bit editing, but with other, ungouped points. Secondly, whereas my first design performed an automatic grouping as a matter of course (the user had no choice of grouping objects explicitly without leaving the bit editor, and no choice of which objects to group), the need to perform automatic grouping in the present design can be thought of as an error condition: points can exist independently only in the bit editor, but the user has attempted to leave the bit editor with some still ungrouped. For this reason, the condition brings up a dialog box warning the user that some points are ungrouped. The user may then choose to have the application automatically group them, or he may cancel the closing of the bit editor. If he cancels, he may go collect the points he wishes to group together on his own, then attempt to leave again.

Other Object Manipulation

Other objects may also be manipulated in the bit editor. Lines, ellipses, circles, and composites may be created, selected, and resized. Selected objects display their handles, just as in zoomed-out mode. Unlike the zoom mode of some other drawing packages, though, the handles in my implementation do not enlarge at this higher magnification. When handles enlarge, they are not only much larger than is necessary for grabbing them, but they interfere with the user's ability to grasp the object itself. Conceptually, too, they should not enlarge. It is not, after all, the handles which the user is attempting to see in greater detail; the handles are a tool, not part of the drawing plane. My handles are always four pixels square. They are distinguishable from points in the bit editor, and fit neatly inside the point, but are large enough to select when stretching is the goal.

Stretching of Objects

I also needed to address the issue of the stretching of objects. When composite objects are stretched, their component objects maintain positions and sizes relative to each other and to the stationary point. These constraints are relatively easy to implement in an object-oriented language like Smalltalk, where all the objects in the composite can be sent similar size- and position-setting messages.

Stretching of Freeform objects

A more challenging problem, once freeform objects are perceived as collections of points, is deciding what ought to happen when a freeform object is stretched. I made the decision, based on mathematical principles, that points should not be stretchable. Points in geometry have no inherent size, just position. Traditional graphics packages, though, allow freeform objects to be

stretched, and users have come to expect this functionality. I needed to decide what stretching meant, both theoretically, and from a user perspective.

My solution -- stretching by multiplying

My decision was to conform to the traditional interface (allowing freeform objects to stretch) without allowing individual points to stretch. Instead, they multiply (or, if the freeform object is made smaller, some die off). If the user were to create a freeform object that is two points square, he would see four points in the bit editor. If he were then to stretch that object from its corner to twice its size and examine it again, he would see sixteen points. This allows me to use traditional methods for sizing bitmaps without abandoning the freeform-as-collection-of-points model.

Other Solutions

Stretching by dispersion

There are certainly other solutions to this problem. I considered allowing two methods of stretching freeform objects: the one above, and a stretch-by-dispersion, in which the number of points stays the same, but they spread out from each other. The problem I anticipated with such a method was that if an original freeform object was made *smaller*, there would be no way to represent, in bitmap form, points that ended up overlapping each other. Instead, those points would be lost. When the object was stretched again, there would be fewer points to be dispersed. So, while there would be two methods of stretching an object, there would be only one method of shrinking it: by losing a number of points.

Stretchable points

I could also have chosen to include stretchable points, or allowed the user to explicitly assign sizes to points just as he can explicitly assign lines different line widths. Although line widths are not consistent with the

mathematical model (lines are, theoretically, infinitely thin), all object-oriented graphics packages allow the user to adjust a line's width attribute. Therefore, I could have justified point sizes with user-interface consistency arguments: that points could be thought of as lines whose beginning and ending points are the same, so should, like lines, have a size attribute. I did not make this decision because of the overhead consideration. Once points take on any attributes of their own, they must be maintained as separate objects (not as parts of freeform objects) outside of the bit editor.

Another reason for rejecting stretchable points is that it would be unclear to the user the difference between points and filled rectangles. They would be functionally the same, and it would be difficult to represent them differently within the bit editor. If the user really wants a point that will get bigger (not just multiply in number), he can create a one-pixel square rectangle (easiest in the bit editor). This is a rather tedious process, however (since creating each rectangle is a click-drag-release process), with unclear benefit.

Disallow stretching

Finally, I could have insisted that freeform objects not be allowed to stretch. This seemed to be the worst alternative, especially for the bit-editing of non-freeform objects. If, for example, the user places a collection of white points over part of the edge of a circle and groups them all together (so he has a freeform object and a circle grouped), chances are that he would want the freeform object to stretch along with the circle when he resizes the group. The freeform object will then cover up the same amount of the circle. In theory, all of the components of a true composite object should maintain relative positions and sizes.

Composite Objects

The implementation of composite objects, as it turned out, was rather trivial due, predictably, to the object-oriented implementation. A composite object is able to respond to many of the messages sent to more primitive objects by simply stepping through its list of sub-objects and passing the message on. Polymorphism is a boon here. There is no need to find out what the objects are before sending them the messages, since they all respond to a common subset of messages. For example, when a composite object is asked to display itself, it does so by telling all of its sub-objects to display.

The stretching of composites is somewhat more involved, since all the objects must maintain certain constraints relative to the whole. When moved, for example, they cannot all move their origins to the same point, but must move to points relative to the new origin of the composite. With stretching, the sub-objects must obey two constraints relative to the whole: position and size. The position in this case is the center position, and the size is based on the original proportion to the whole prior to stretching.

Circle objects presented a unique problem when stretching them as components of a composite. I used, primarily, MacDraw and Superpaint as my models for object behavior, but neither of these packages separates circle objects from ellipses in any fundamental way. Circles are ellipses which happen to be round: a temporary condition initiated by creating or stretching an ellipse while the shift key is depressed. If the user happens to stretch a circle without the shift key down, the circle will become elliptical.

In my package, though, CircleObject is a subclass of the EllipseObject. With this implementation, they are able to maintain their roundness under any conditions, including stretching. When a circle is stretched alone, its new

diameter is derived from the minimum of the width and height of its new bounding box.

When any object is stretched as part of a composite, however, its new extent is derived proportional to the new extent of the entire composite. This new extent may be square but is more likely to be rectangular. I needed to make a decision, then, what combination of width and height of the new extent should determine the diameter of the new circle within the composite. I attempted to take the minimum as I had when circles stretched independently. When I did, the circles became successively smaller with each transformation. I also tried averaging the width and height. In that case, they became successively larger. Finally I used only one dimension (somewhat arbitrarily, width) to determine the new radius, and ignored the other. As a result, circles which are parts of composites will change their radii if the stretching occurs in the horizontal direction, but will maintain their old radii when stretched vertically. This approach has the advantage of being reversible: a stretch to one position and back again will reproduce the same configuration of shapes as in the original.

Class Implementation

Hierarchy, Inter-module Communication, and Data Flow

Figures 8a and 8b are diagrams of my system hierarchy. Bolded classes are classes that were in place before I started coding. All other classes are classes I created. Events are generated through user interaction with the ObjectOrientedDrawing, either through menu commands, through page changes, or through direct manipulation of the graphical objects. These events are fielded by the ObjectOrientedDrawing and forwarded, if necessary, to the relevant graphical object or objects. Data is, of course, maintained within each object, so data flow is simplified. It is primarily a matter of ObjectOrientedDrawing setting or requesting the states of instance variables.

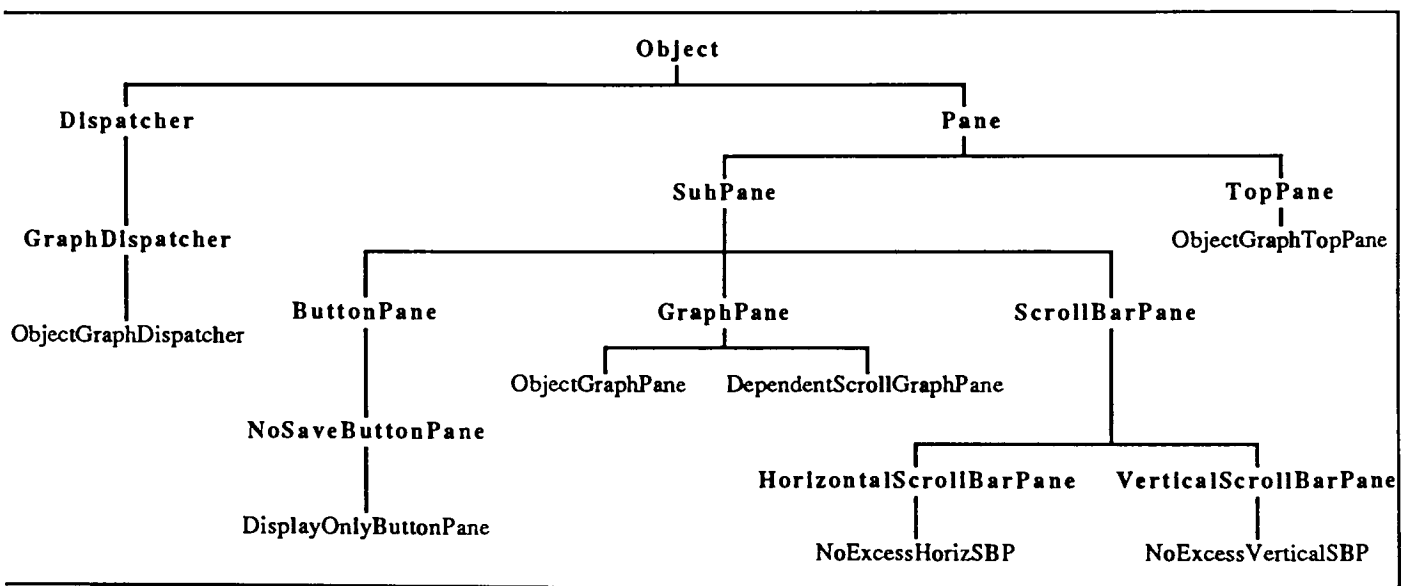


Figure 8a -- Panes and Dispatchers

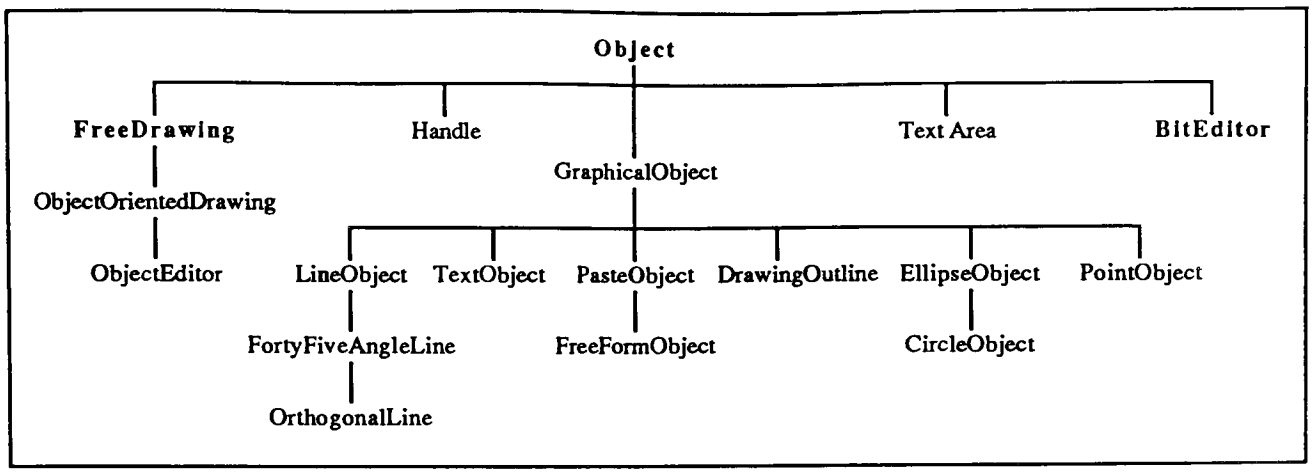


Figure 8b -- ObjectOrientedDrawing, GraphicalObjects, Others

Classes

ObjectOrientedDrawing

Just as Fuchsberger and Krasemann did, I began my implementation by inheriting from the FreeDrawing class. The subclass, ObjectOrientedDrawing, contains the bulk of the code for creating, manipulating, and storing the graphical objects. It also serves as the model for my MPD interface. As stated earlier, however, I gained less than they did from the inheritance. I ended up inheriting only two methods without overriding them: one which converts a position to its scrolled position, and one which distributes mouseDown events based on the current state of the drawing (line, circle, etc). I did borrow many of FreeDrawing's ideas, though, and overrode many of its methods.

The ObjectOrientedDrawing's main data structure is a list of graphical objects. ObjectOrientedDrawing uses this list for adding newly created objects, deleting removed objects, rearranging the order of objects on bringToFront or sendToBack, and searching for the currently selected object. ObjectOrientedDrawing also contains a list of pages, which are Forms, and it knows the current page. Since each graphical object knows the form it is on, it also implicitly knows its page.

GraphicalObject

Inheritance helped me more in the implementation of the graphical objects. The objects themselves are subclasses of an abstract class, GraphicalObject. Its subclasses specialize its abstract behaviors for displaying, finding their bounding boxes, changing their origins etc. GraphicalObjects have their own sets of Handles (see "Supporting Classes" below), and are responsible for creating the appropriate number of handles and placing them in the appropriate locations. They also know whether those handles are currently being displayed (whether the object is selected). They are able to erase themselves, copy themselves to the clipboard, and save themselves out to a file.

Each GraphicalObject also holds the unique instance variables it needs for its mathematical definition. LineObjects, for example, hold their two endpoints. EllipseObjects know their horizontal radii and aspect ratios. CircleObjects are ellipse objects with aspect ratios of 1. (Implementing circles was a pleasure, since they inherited so much functionality from ellipses.) CompositeObjects have a list of all their sub-objects. PointObjects are implemented to exist only in the bit editor; each stores its location and color.

PasteObjects differ from their subclass, FreeFormObjects, primarily in the way in which they are created. PasteObjects are created by copying a rectangular area out of the bitmap; freeforms are created either with the drawing tool, or by grouping points together.

For the implementation of TextObjects, I created a TextArea class, which is not a GraphicalObject but a subclass of Object. (This was a design decision I made early. If I were to write it over again, I would probably combine the TextArea and TextObject. Many of the methods in TextObject simply pass the message on to the TextArea it holds.) TextArea has some of the same

functionality of a TextPane, but does not cover an entire pane, only a small rectangular area. Text is restricted to one font and point size per block; variable type faces (bolding, italics, etc.) are not implemented.

Although Figure 8b shows two subclasses of LineObject, I have only written the class definitions of FortyFiveAngleLine and NinetyAngleLine, for future extensions including these objects. They currently have no methods. The more general LineObjects can stretch in any direction; NinetyAngleLines would be orthogonal, and FortyFiveAngleLines would be restricted to maintaining a (negative or positive) 45° angle.

The last GraphicalObject is DrawingOutline, which is an object that simply displays the edges of the drawing space. It cannot be selected, moved, or stretched.

All GraphicalObjects are aware of their "hosts" in the same way that a text pane is aware of its model. In this case, the host is either an ObjectOrientedDrawing instance or an ObjectEditor instance. The class of the host determines, among other things, how the object will display. In the ObjectEditor, for example, it must display twice: once in the inset window at normal scale, and once at eight-times its normal magnification.

ObjectEditor

FreeDrawing's BitEditor

From an implementation standpoint, I needed to resolve just what the bit editor should be. In the FreeDrawing implementation from which I borrowed, the BitEditor class was a subclass of Object and was quite independent from the FreeDrawing class, both in implementation and in functionality. The user would choose to bit edit a rectangular area of the drawing, and a bit editor would come up with that area expanded. This bit editing window looked quite different from the original drawing window. There was only one tool, which

would either create or remove black bits while the mouse button was pressed, depending on whether the original click was on a black bit or not. (It reversed this first bit, then remained in creation or removal mode.) The window had dotted lines to show the bit boundaries, buttons for exiting and saving the bit editor, and no extra menus. It was obviously a different type of object from the drawing window.

ObjectEditor as a Subclass of BitEditor

I originally attempted to subclass the BitEditor class with my ObjectEditor class, but soon it became clear that the functionality of my bit editor (the object editor) was closer to the functionality of my ObjectOrientedDrawing class than it was to the original BitEditor. I saw that I would be rewriting much of the ObjectOrientedDrawing's code if I made the ObjectEditor a separate class. Instead, the ObjectEditor class inherits from ObjectOrientedDrawing.

...As an Instance of ObjectOrientedDrawing

Conceptually, too, the ObjectEditor is a *kind of* ObjectOrientedDrawing. It includes the same set of drawing tools, and the objects are being created and manipulated in the same ways. Objects can, for example, be selected and stretched or moved. They are so similar, in fact, that I toyed with the idea that they should be instances of the same class, with only the size of their bits differing (a difference of one instance variable). In that case, they could be thought of merely as different views on the same sets of objects.

...As a Subclass of ObjectOrientedDrawing

I decided, however, that the ObjectEditor is different enough from an ObjectOrientedDrawing to justify the inheritance. It has, for example, an inset window to show the actual size of the drawing. It has a tool palette (something that could actually be added to the ObjectOrientedDrawing eventually). It also has some different functionality. It is, as stated earlier, the only place where

point objects can be manipulated independently. It also reacts differently to closing, since it must check for ungrouped point objects; and differently to grouping, since it must group points into freeform objects, and check to see that all the points in a grouping are the same color. Unlike the ObjectOrientedDrawing, it has no notion of pages. Finally, it is modal: while it is open, other windows may not be selected. This avoids multiple concurrent updates to the same ObjectOrientedDrawing window.

Supporting Classes

Handles

I created a Handle class (subclass of Form) to implement the small black handles which indicate whether an object is selected. They are also used to stretch or reshape an object, so they must be able to reverse themselves when selected, report whether or not they contain a given point, and reposition themselves to a new location.

Panes and Dispatchers

I had to create my own dispatcher to change scrolling and scroll-bar paging distances, and to keep track of whether the drawing had been modified. I called it ObjectGraphDispatcher. I then needed an ObjectGraphPane which knows that its default dispatcher class is ObjectGraphDispatcher, and asks its model to add specific menus. I also needed an ObjectGraphTopPane to change the default File menu, and manage the saving of drawings to my own file format.

To implement the ObjectEditor, I needed a GraphPane whose scrolling could depend on the scrolling of another pane. This would assure that both the inset pane and the bit pane were displaying the same portion of the drawing. This class is DependentScrollGraphPane. It is added with the inset pane (an ObjectGraphPane) as its model, and then responds to its model's update message

whenever scrolling is involved. The DependentScrollGraphPane also uses ObjectGraphDispatcher as its default dispatcher class.

I needed to modify scroll bar panes to keep them from ever showing more of the drawing than actually exists. They are NoExcessHorizontalSBP and NoExcessVerticalSBP.

The only other pane class I created for the thesis was a DisplayOnlyButtonPane, which cannot be selected like any other button pane, but displays relevant text. I use it to show the page number of the page which the user is currently viewing.

New Methods for Existing Classes

I also added methods to a smattering of pre-existing classes to enhance their functionality for my needs. For example, I added Bressenham line, circle, and ellipse algorithms to the pen class, so that I could display these shapes in the bit editor with stair-step-like edges. (Pen's original line and ellipse methods are written as primitives, and the source code is not available.)

I added methods to the Dialog class, to provide access to the dialog boxes I created with a Macintosh resource editor; and to the SFReply class to provide access to my "Save Changes?" dialog. Other enhanced classes were Form (for scaling PasteObjects), Point (for real-number division), and Rectangle (for changing specific x- and y- values of its origin and corner).

I added about a dozen methods to the class TextSelection, for showing selections on a particular form, not just the window. (In retrospect, I probably should have created a subclass for these methods, since there were so many, and since their functionality was related.) I also added a method to the Smalltalk class StringModel to allow it to display on a particular Form, not just in the front-most window.

Resources

In addition to the dialog boxes for saving changes and alerting the user to ungrouped points in the bit editor; I created a set of cursor resources (using the Macintosh resource editor) which indicate to the user which drawing state he is currently in. The cursor may be in the shape of a circle, an ellipse, a cross-hair, a pencil, etc. The management of these cursors is implemented by my DrawingCursorManager class, a subclass of CursorManager.

Conclusions and Future Extensions

It is possible to combine object-oriented and bit-mapped interfaces into a single coherent application. While the solution for this integration was not derived overnight, the use of object-oriented design principles sped the development of a complex graphical user interface, and, I believe, provided fresh insight into the problem of representing bit-mapped objects. Because Smalltalk enforces the notion that every element in the system is an object, the Smalltalk developer is forced to begin designing his solution purely in terms of objects. This mind-set allowed me to view the point as no other graphics package has presented it: as a unique graphical entity (just as it is in formal geometry) available to the user as a graphical tool. With the existence of a point object, the remainder of the problem became simply a matter of designing an interface which would allow the user to effectively manipulate these points. As a result, users of my package are able to enjoy the benefits of both bit-mapped and object-oriented editors without ever abandoning an environment in which every graphical element is an object.

Current Features

Besides combining bit-mapped and object-oriented graphics, my implementation includes a subset of the standard draw and paint-package

features. The check marks (√) in the tables in Appendix A indicate the features I have implemented. My package is also includes a feature which is not in the tables, but is a desirable one for Macintosh programs: the ability to cut and paste bit-mapped graphics between mine and other applications using the Macintosh clipboard.

For detailed explanations of how my application's features are to be used, see the user manual in Appendix B. It shows what inputs are expected, including keyboard and menu commands, and tells what graphical output the user can expect.

Future Extensions

Any of the other features in the tables could be considered as future extensions to this thesis. In addition, there are some of which I found no mention that would enhance the application. One is the ability to draw a reference line (i.e. a line that would not show up on the final drawing) and mirror an object across the line in one direction or another. This differs somewhat from the "flipping" feature found in many packages, where the whole object is flipped either horizontally or vertically. What I envision would only mirror the part of the object "in front of" the mirror and delete the part of the object "behind" the mirror. In this way, users would be able to create half of an object and make it completely symmetrical along an axis. At first, this could be implemented over a horizontal or vertical reference line (if any object could be rotated, this is all that is really necessary).

Another feature not in the tables could be the inclusion of arrows on cubic curves, typical of those often drawn freehand to point out parts of an illustration.

I have already put provisions in place for stretching objects with the shift key depressed. This could allow alternative methods of stretching. Circles and

ellipses, for example, which currently stretch by staying stationary at the corner opposite the one selected, could have the option of stretching from the center. As another example, a line might maintain its current slope when it is stretched with the shift key down.

Currently, only lines, paste objects, and text objects are able to store themselves to files. The object begins by storing the name of its class. A LineObject then stores its beginning and ending points, the size of its pen, and the pen's mask (for dotted lines). A TextObject stores its font and point size, location, and string. A PasteObject stores its bitmap, its size and location, and its angle of rotation. Other objects should certainly be added to this list. The current file format is my own and, except for the bits in a paste object, is purely textual. Another future extension might be to offer a variety of standard file formats, like TIFF and PICT.

Ideally, the application should be launchable by double-clicking on its icon at the Macintosh Finder level; it should not be necessary to evaluate a Smalltalk expression to start the application. According to Digitalk, this feature will be possible in the next version of Smalltalk/V Mac.

Undo should work for more than just cut and paste operations. Some other operations that should be reversible are: moving, stretching, clearing, typing, deleting, and font changes. Also, font and point size information should be available on its own menu. It is not really appropriate for the Window menu.

Two operations that would be easily added are bit editing at more than one magnification (it is currently eight times the original); and a Select All command for a given page. The ObjectEditor class returns its size whenever it is requested -- it is not hard-coded. Therefore, an ObjectEditor could be told to open at a given level of magnification. Select All would be simple because all the objects on the current page can be told to turn their handles on.

Finally, pick sensitivity is currently rather crude for some objects. Most inherit from GraphicalObject the method **contains:** which reports whether or not a point is close enough to the object to be considered an attempt at picking it. This method is implemented by testing whether or not the point is inside the object's bounding box. The one exception is the line, which calculates its slope and the slopes of the line perpendicular to it which intersects the point, then measures distance. A similar method could be implemented for EllipseObjects (and, through inheritance, circles), by calculating the slope of the tangent near the selected point.

Implementation Alternatives

Due to the fact that I began writing this program when I was first learning Smalltalk/V, there are a few things I would have done differently, had I started with my current level of understanding. One involves the updating of the drawing when an object is moved. This is currently a rather complicated process, involving the sending of a number of **self changed** messages. It is particularly involved when an entire collection of objects is moved at once. It seems to me that the process need not be quite as complex as it is.

Something which might simplify the above process would be a modification of the way in which collections of objects are treated. There are two ways of moving objects together. One is by grouping the objects; they then become a composite object, and any number of transformations can be performed on the composite. The other is by simply selecting many objects at a time. Currently, this collection can be moved; then, when the user clicks outside of any of the objects, the individual objects lose their association with one another.

The way I have implemented these groups of objects is to treat a selected collection differently from an individually selected object. To move them, a separate method in ObjectOrientedDrawing is invoked. Consequently, the

objects in the collection cannot be stretched together as a composite object's sub-objects can: this would have involved the writing of another special case.

A more elegant solution would be to create a subclass of CompositeObject, called TemporaryComposite. When objects are grouped into a composite, they maintain their front-to-back ordering relative to each other, but lose it relative to the objects not in the composite. (The composite itself becomes the new front-most object.) The objects in a TemporaryComposite should not lose their orderings, however. Furthermore, unlike a CompositeObject, whose objects are all known when the object is created, new objects could be added to a TemporaryComposite every time the user selects a new object with the shift key down. Likewise, the TemporaryComposite should go away when the user clicks outside of any of its members, unlike the CompositeObject, which is explicitly ungrouped.

As a subclass of CompositeObject, though, TemporaryComposites would inherit all the behavior for updating their positions and stretching. No special methods would have to be implemented in ObjectOrientedDrawing for handling collections of selected objects.

The last change in implementation that I would make would be to give ObjectOrientedDrawing a default scale. I found that graphical objects were often querying their hosts to find out if they were "**aKindOf:** ObjectEditor" simply to learn whether to multiply some value by the host's scale. If ObjectOrientedDrawings also responded to the message **scale** by returning 1, many (but not all) of these queries would be unnecessary.

Bibliography

Barker, D. "Studio/8: The Best Paint Yet." *Byte*. March, 1989. p. MAC5.

Bowman, William and Bob Flegal. "Toolbox: A Smalltalk Illustration System." *Byte*. August, 1981. pp. 369-376.

Burns, Diane and S. Venit. "The Fine Points of Illustrator." *Publish!* July, 1987. pp. 78-82.

Busch, David D. *Secrets of MacPaint, MacWrite, and MacDraw*. Little, Brown, and Company. Boston. 1986.

Cline, Craig E. and Gregory P. Stern. "Target Software's Scoop." *The Seybold Report on Desktop Publishing*. March, 1988. pp. 13-22.

Cox, Carl and Duff Caldeway. "Six State-of-the-art Mac Graphics Packages." *Computer Language*. October, 1987. pp. 93-106.

Deutsch, L. Peter. "The Past, Present and Future of Smalltalk." *ECOOP '89, Proceedings of the Third European Conference on Object-Oriented Programming, 1989*. Steven Cook, editor. pp. 73-87.

Dodani, Mahesh H., Charles E. Hughes, and J. Michael Moshell. "Separation of Powers." *Byte*. March, 1989. pp. 255-262.

Fenton, Erfert. "SuperPaint 2.0." *MacWorld*. August, 1989. p.169.

Fenton, Erfert. "The Big Match: Illustrator 88 vs. Freehand." *MacWorld*. February, 1989. pp. 180-187.

Fuchsberger, Rudolf and Hartmut Krasemann. "Object-Oriented Drawing -- Reusing the FreeDrawing Class." *Journal of Object-Oriented Programming*. May/June, 1990. pp. 9-16.

Giguere, Eric. "Skin and Bones." *Computer Language*. April, 1990. pp. 43-59.

Goldberg, Adele and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Publishing Company. Reading, Mass.1989.

Guzelimian, Vahé and Adrian Mello. "Drafting's New Compass." *MacWorld*. January, 1986. pp. 92-95.

Haaland, Kevin and Dave Thomas. "SmallScript: A User Programmable Framework Based on Smalltalk and PostScript." *Proceedings, Graphics Interface, '89*. pp. 55-61.

Heid, Jim. "Getting Started with Macintosh Graphics." *MacWorld*. August, 1989. pp. 193-202.

Hopgood, F.R.A., D.A. Duce, J.R. Gallop, and D.C. Sutcliffe. *Introduction to the Graphical Kernel System (GKS)*. Academic Press. London. 1983.

Ingalls, Daniel H.H. "The Smalltalk Graphics Kernel." *Byte*. August, 1981. pp. 168-194.

Jones, Mark. *Human-Computer Interaction: A Design Guide*. Education Technology Publications, 1989.

Kay, Alan. "Smalltalk." *Methodology of Interaction. Proceedings of the IFIP Workshop on Methodology of Interaction*. R.A. Guedi, P.J.W. Ten Hagen, F.R.A. HopGood, H.A. Tucker, and D.A. Puce, editors. 1980. pp. 7-11.

Kerlow, Isaac Victor. "Fine Lines: The PC as Illustrator." *PC Magazine*. June 27, 1989. pp. 129-170.

Kirsh, Laurence. "A Graphics Workhorse." *MacWorld*. January, 1988. pp. 157-158.

Kleinman, Lisa. "Illustrator: Adobe Throws a Curve." *Personal Computing*. August, 1989. pp. 158-159.

Kramer, Axel. "IconMaker: Interactive User Interface Design." *1984 IEEE Computer Society Workshop on Visual Languages.*, pp. 192-198.

LaLonde, Wilf and John Pugh. "Pluggable Tiling Windows." *Journal of Object-Oriented Programming*, September/October, 1989. pp. 57-66.

LaLonde, Wilf and John Pugh. "Windows, Panes and Events in Smalltalk/V PM." *Journal of Object-Oriented Programming*, February, 1991. pp. 57-66.

London, Ralph L. and Robert A. Duisberg. "Animating Programs Using Smalltalk." *Computer*. August, 1985. pp. 61-71.

Lorensen, W., M. Barry, P. McLachlan, and B. Yamrom. "An Object-Oriented Graphics Animation System." *Object-Oriented Geometric Modeling and Rendering. 14th Annual Conference on Computer Graphics and Interactive Techniques*. July, 1987.

McKinstry, Steve. "Draw, Pardner." *MacWorld*. August, 1989. pp. 140-147.

Meyers, Bertrand. *Object-oriented Software Construction*. Prentice Hall. New York. 1988.

Mohageg, Michael F. "Differences in Performance and Preference for Object-Oriented vs. Bit-mapped Graphics Interfaces." *Proceedings of the Human Factors Society 33rd Annual Meeting*. 1989. pp. 385-389.

Mohamed, Ramzen. "An Experiment in Algorithm Animation using Smalltalk on a Macintosh." *Graphics Forum*. May, 1987. pp. 151-155.

Pogue, David. "Two-Tone News." *MacWorld*. December, 1989. pp. 154-161.

Robinson, Phillip. "Drawing, Drafting, and Design." *Byte*. July, 1987. pp. 251-256.

Roth, Steve. "Well-Rounded Drawing." *PCWorld.* July, 1989. pp. 164-174.

Seiter, Charles. "Smalltalk/V Mac 1.0." *MacWorld.* August, 1989. pp. 190-192.

Shneiderman, Ben. *Designing the User Interface.* Addison-Wesley Publishing Company, 1987.

Shuey David, David Bailey and Thomas P. Morrissey. *PHIGS.*

Smith, David, Charles Irby, Ralph Kimball, Bill Verplank and Eric Harslem. "Designing the Star User Interface." *Byte.* April, 1982.

Tazelaar, Jane Morrill. "Object-Oriented Programming." *Byte.* March, 1989. p. 228.

Texas Instruments Incorporated Data Systems Group. *Explorer Tools and Utilities.* Texas Instruments Incorporated. Austin, Texas. 1987.

Texas Instruments Incorporated Data Systems Group. *Explorer Window System Reference.* Texas Instruments Incorporated. Austin, Texas. 1987.

Thomas, Dave. "What's in an Object?" *Byte.* March, 1989. pp. 231-240.

Thompson, Tom. "The Next Step." *Byte.* March, 1989. pp. 265-269.

Udell, John. "Bridging Troubled Waters." *Byte.* April, 1990. pp. 225-230.

Urlocker, Zack. "Object-Oriented Programming for Windows." *Byte.* May, 1990. pp. 287-294.

Wegner, Peter. "Learning the Language." *Byte.* March, 1989. pp. 245-253.

Wisskirchen, Dr. Peter, and Erich Rome. "Object-Oriented Graphics." *ACM/SIGGRAPH Course Notes.* 1988.

Wisskirchen, Peter. "Geo: Graphics System with Editable Objects." *Advanced Computer Graphics. Proceedings of Computer Graphics, Tokyo, '86.* pp. 172-179.

Wisskirchen, Peter. Towards Object-Oriented Graphics Standards. *Comput. and Graphics.* 1986. vol. 10, no. 7. pp. 183-187.

Yankelovich, Nicole, Bernard J. Haan, and Steven M. Drucker. "Connections in Context: The Intermedia System." Institute for Research in Information and Scholarship, Brown University, 1988.

Appendix A: Feature Tables

Table 1* -- Drawing Package Features

	General		Object Manipulation/Editing
	On-line help		Multiple copies of selected object
	Customize menus/toolbox		Snap to grid
√	Open multiple files		Curves scaled with handles
	3-D		Curves reshaped with control points
	Color		Polygons/lines reshaped with control points
			Select/move multiple points
	Drawing Tools/Object creation		Tangents to curves visible
√	Straight lines		Can insert points into or delete points from a curve or polygon
	Orthogonal lines		Automatic curve smoothing/unsmoothing
√	Selectable line endings		Polylines can be converted to curves
	Invisible lines		Attributes can be copied from one object to the next
	Standard arrowheads		Paths renewable
	Customizable arrowheads		Rectangles divisible into lines
	Rectangles		Paths divisible
	Squares		Combine curves, polylines
	Rounded corners for rectangles		Line patterns
	User-controllable rounded corner		Dashed-line editor
√	Circle		Skewing or shearing
√	Ellipse		Flipping, horizontally and vertically
	Arcs		Flipping along any axis
	Circle segments	√	Rotations by 90°
√	Freehand drawing	√	Rotations by 1°
	Automatic closure, curved paths		Rotations around a movable origin
	Polygons		Gray percentage fills
	Polylines		Pattern fills
	Regular multigons		Graduated fills
	Linear grates		Resizable by original proportions
	Logarithmic grates		Resizable to square proportions
	Starburst		Scalable line thickness
	Triangles		Freehand pencil flips to eraser w/ a keystroke
	User-definable object types		Calligraphic pen shapes
	Nested concentric shapes		Distribute/blend tool for shapes or colors
	Shadows		Use objects as masks
	Spline curves		Background pictures available for tracing
	Bezier curves		Auto-tracing
	User editable patterns		Layers available
	Draw-from-center option		Layers namable
√	Bitmaps may be included in the image		Layers can be reordered
			Layers can be printed individually
	Text	√	Move to back or front
√	Editable on-screen	√	Group/ungroup objects
	Can be bound to a curve	√	Multiple selection
	Manual kerning		Objects can be aligned to the page

* Table 1 was compiled from the following sources: [FLAVORS GRAPHICS EDITOR REFERENCE] [Guzelimian and Mello, 1986] [Kirsh,1988] [Heid, 1989] [Roth, 1989] [Kleinman, 1989] [Fenton, 1989a] [Fenton, 1989b] [Cline and Stern, 1988] [Kerlow, 1989] [Burns and Venit,1987] [Robinson, 1987] [Cox and Caldeway, 1987] [McKinstry, 1989] [Yankelovich et. al., 1988] [Busch, 1986].

Table 1 -- Drawing Package Features, continued

	Text, continued		Object Manipulation/Editing, cont.
	Automatic kerning		Objects can be aligned to each other
	Adjustable letterspacing		Automatic scrolling with selection tools
	Letterforms re-shapable		Automatic scrolling with drawing tools
	Fill or outline		Arrow key movement
	Use as mask		Arrow key nudging
	Multiple fonts, sizes, styles per block		Tab key for sequential selection
	Word wrapping available	√	Undo
	Text can be rotated, flipped, scaled, skewed		Multiple or unlimited undo
	Text blocks can be merged		Lock objects into place
	Reversible type		
	Display		**Measurements**
	Zoom feature		User-definable units
	User-definable zoom		User-definable grid
	Auto-dimensioning		User-definable page size
	Status box with alphanumeric information	√	Multiple pages
	ViewFinder for rapid, direct movement		Rulers as objects
	Status line		Rulers on margin with crosshairs
			Customizable rulers

Table 2** -- Paint Package Features

	Object Manipulation/Editing		Paint Features, continued
	Lasso-click to select small portions		Blend tools
	Slant selected area		Smudge tools
	Rotate selected area		Paint with patterns
	Create vanishing-point perspective on selected area		Watercolor effects
			Other Features
	Painting Features	√	Real text
	Gradient fill		Autoscroll at window edge
	Radial fill		Movable/resizable inset window when zoomed
	Variable airbrush	√	Revert-to-saved eraser
	Paint-bucket masking		Movable tool palette
	Editable brush		Ability to hide tool palette
	Editable patterns	√	Fat-bits bit editor
	Color sampler	√	Eraser

** Table 2 was compiled from the following sources: [Pogue, 1989] [Roth, 1989] [Heid, 1989] [Barker,1989] [Busch, 1986] [Cox and Caldeway, 1987].

Appendix B: User Documentation

Launching the Application

The application can be started up only from inside Smalltalk/V. To "launch" the application, you may evaluate an expression to either create a new drawing or open an old one. To create a new drawing, evaluate the following Smalltalk expression with **Do it** (⌘D) under the **Smalltalk** menu:

```
ObjectOrientedDrawing new open.
```

You will get a drawing window labeled "**Untitled**" with a dotted-line border around the drawing area. (See figure B-1.)

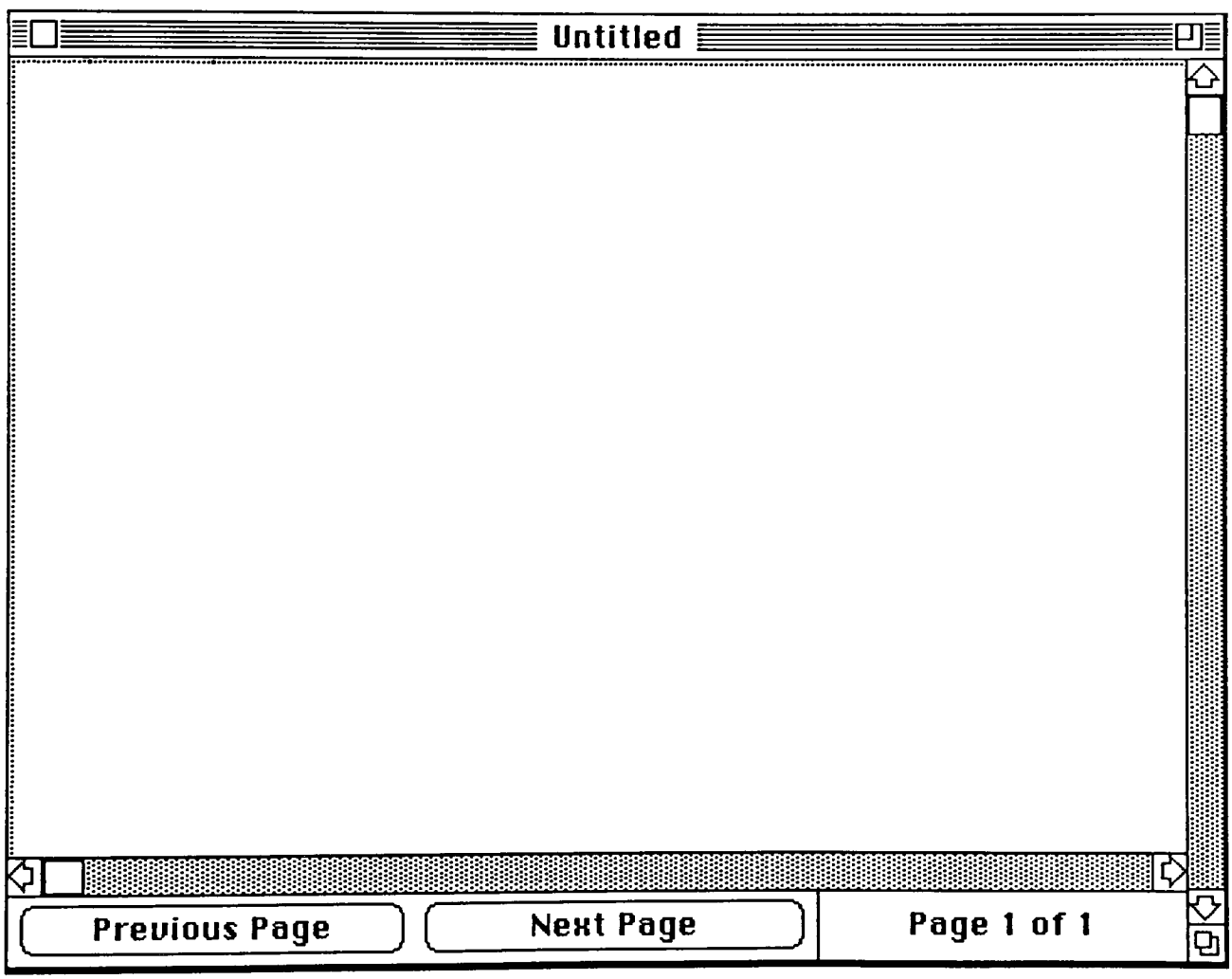


Figure B-1 -- A new drawing

A previous drawing may be opened by evaluating:

ObjectOrientedDrawing new openOld.

In this case, you will get a dialog box prompting you for the file name of a drawing that was previously saved. After a file name is chosen, a drawing window will appear whose label is the file name chosen. It will contain all objects previously saved to this drawing, and the dotted line border. If the drawing selected is already open, the window containing that drawing will be brought to the front of the screen.

Once either a new or an existing drawing is open, operations may be executed through the menus. No further Smalltalk commands need be evaluated. The remainder of this manual is constructed as a tour through the menus, with explanations of buttons where most appropriate.

The File Menu

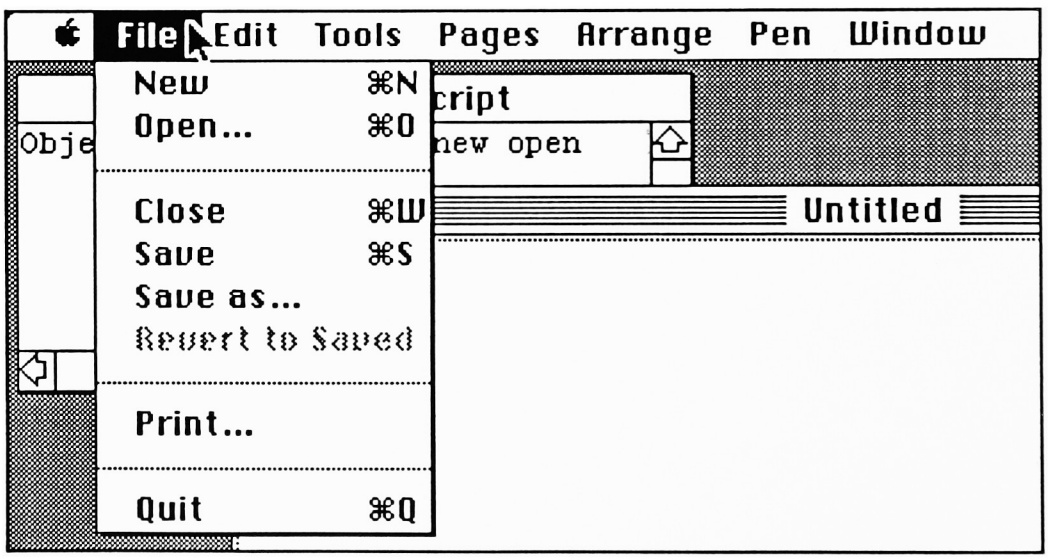


Figure B-2 -- The File Menu

New

Each **New** command will open another untitled drawing containing one page and no objects. (See figure B-1.) This command may also be executed by typing ⌘N.

Open

The **Open** command (⌘O) will open a drawing from any file which is in the file format expected by the application (type 'FORM'). Only files in this format (and folders) will appear in the **Open** dialog box. When a file is opened, it will display the first page and all the objects on that page. If the drawing you select is already open, the window containing that drawing will be brought to the front of the screen. This prevents multiple updates to the same drawing.

Close

The **Close** command (⌘W) will remove the top window. If you have made changes to the drawing in this window, a dialog box will appear asking if you want to save those changes (see figure B-3). Clicking **Yes** (or pressing the return key) will save the changes and close the window. Clicking **No** will close the window without saving the changes. Clicking **Cancel** will neither save the changes nor close the window.

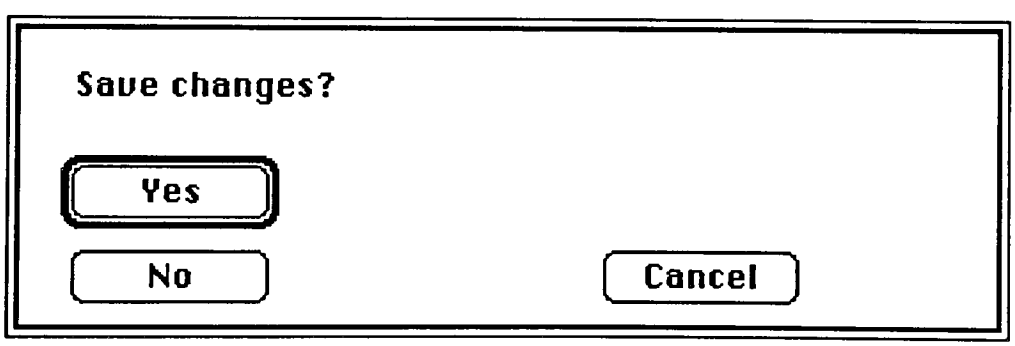


Figure B-3 -- Save Changes Dialog

Save

The **Save** command (⌘S) will store the drawing in its current state out to a file in the application's own file format ('FORM'). If the drawing is still an untitled drawing, a save-file dialog box will appear to allow you to select a folder for the file and type in a file name (see figure B-4). If the name you

type in is already a file name in the current folder, you will be asked whether to overwrite the old file. If you say **No**, you may type in a new name.

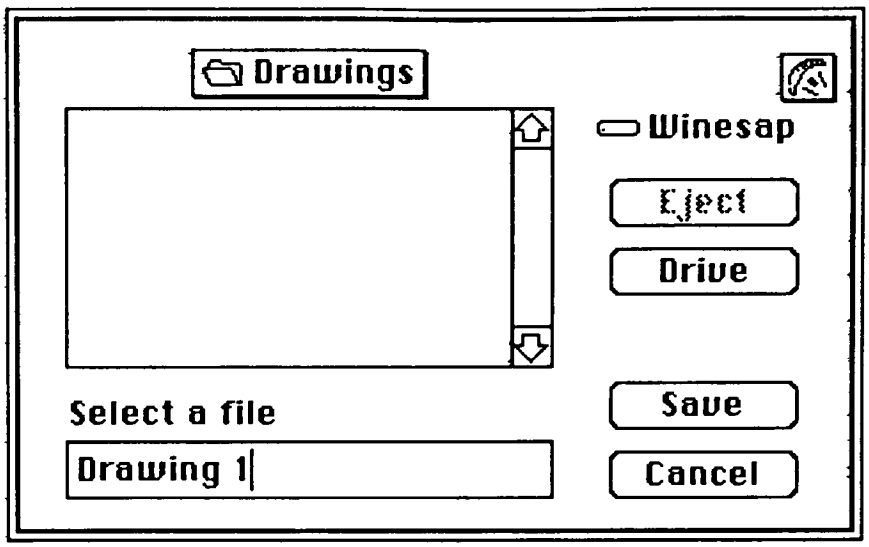


Figure B-4 -- Save File Dialog

The save-file dialog box will also appear if you attempt to close a modified but untitled drawing window and click **Yes** in the Save Changes Dialog (figure B-3).

All previously-named drawings will be saved under their current file names; no dialog box will appear.

Save As...

Drawings that have previously been saved may also be saved under new names by choosing the **Save As...** option under the **File** menu. Again a save-file dialog box will appear (see figure B-4) to allow you to choose a folder and file name. After the save operation, both the drawing's file name and the drawing's label will change to the new file name. (The **Save As...** option is not available for untitled drawings, since the **Save** option for untitled drawings invokes the same dialog box.)

Print

The **Print** command spools the drawing out to a printer. The standard Macintosh print dialog appears to request which pages are to be printed, etc. (See figure B-5.) In this version of the application, only a bit-mapped (72 dots-per-inch resolution) version of the drawing will be printed.

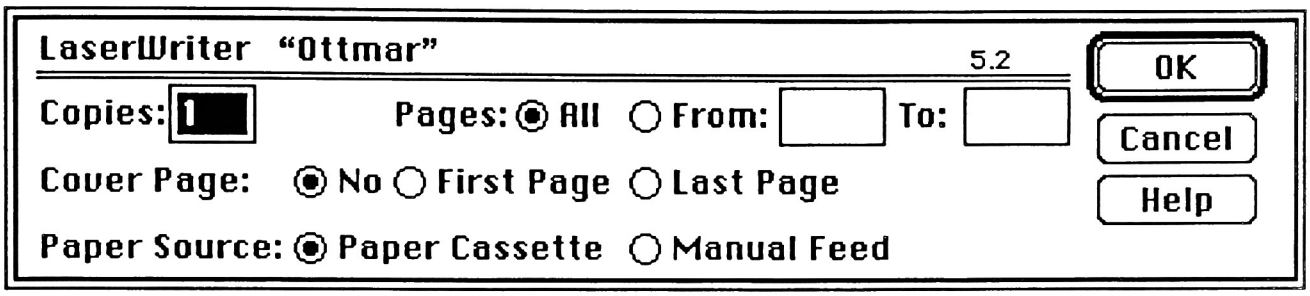


Figure B-5 -- Print Dialog

Quit

If the drawing has been modified since the last save operation, invoking the **Quit** command (⌘Q) initiates a dialog box asking if you want to save your changes, as described above for the **Close** command (figure B-3). It then allows you to leave Smalltalk.

The Edit Menu

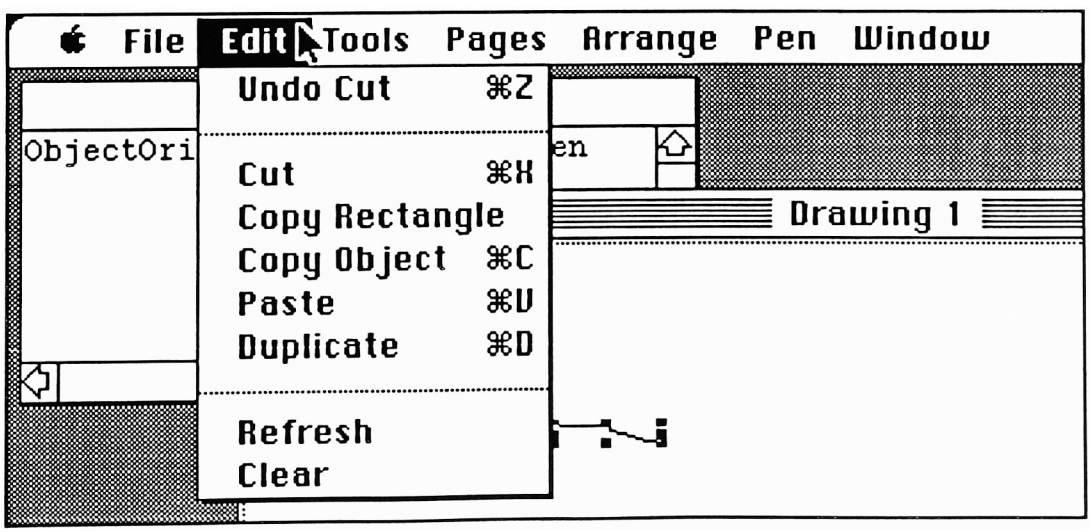


Figure B-6 -- The Edit Menu

Undo

In this version of the application, the **Undo** command (⌘Z) only reverses the effect of cut and paste operations. If the last operation was cut, it does a paste; if it was paste, it does a cut. (The menu reflects this. It will say either **Undo Cut** or **Undo Paste**.) It does not remember the previous location of the cut object, so undoing a cut operation will allow you to paste the object back into the drawing anywhere. If there has not yet been a cut or paste operation, the **Undo** command will be grayed, indicating it is not an available operation.

Cut

Cut (⌘H) works on any object which is selected (has its handles on). The object selected is removed from the drawing, and is placed on the clipboard. Until something else is placed on the clipboard by way of a cut or copy operation, that object may be pasted back into this or another open drawing. It may also be pasted (as a bitmap only, not as an object) into any other Macintosh application which accepts bitmaps.

If there is no object selected, the **Cut** operation will have no effect.

Copy Rectangle

Use the **Copy Rectangle** command to copy to the clipboard any rectangular area of the drawing. After choosing this tool, an upper left-hand (origin) cursor will appear (see figure B-7a). The first place the mouse button is both pressed and released will then be the initial corner of the rectangle to be copied. (This is not necessarily the upper left-hand corner; that depends on which direction the mouse moves next.) The cursor will then change to a lower right-hand (corner) cursor (figure B-7b). A dotted-line rectangle will rubber-band to the next location until the mouse button is pressed and released (figure B-7c). The rectangular area of the drawing which was outlined will be copied to the clipboard as a bitmap.

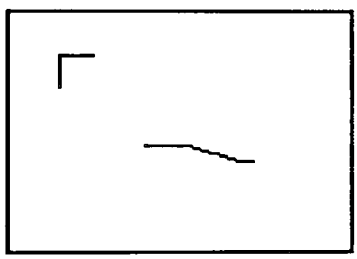


Figure B-7a

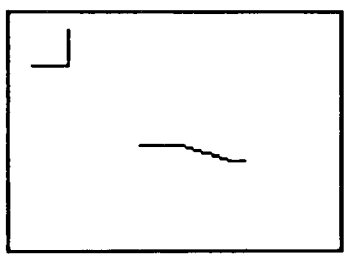


Figure B-7b

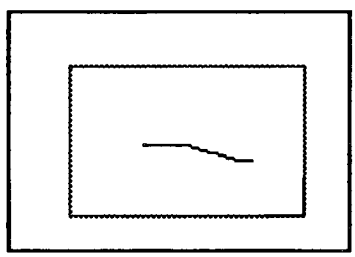


Figure B-7c

Copy Object

Copy Object (⌘C), like **Cut**, copies any selected object to the clipboard; but unlike **Cut**, does not remove the object from the drawing. From the clipboard, it may be pasted on this or any other open drawing (or another application). If no object is selected, **Copy Object** has no effect.

Paste

Anything on the clipboard, whether text or graphics, may be pasted into the drawing. (For a detailed description of how text may be pasted, see Text below.) When you choose **Paste** (⌘U) from the **Edit** menu, a dotted-line rectangle of the size of the object (see figure B-8a) will track your mouse as

far as the limitations of either the window or the drawing area -- whichever is smaller. When you click on the mouse, the object will be placed on the drawing at that mouse location. It's "handles" (eight small black squares surrounding the object) will be visible, indicating that the object is selected. (See figure B-8b.) This means that you have been placed in pointer mode (see **Pointer** below). It also means that the object can be moved or stretched (again, see **Pointer**), cut, copied as an object, or duplicated.

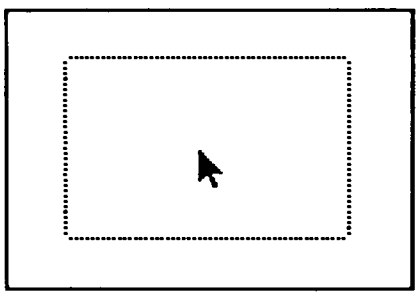


Figure B-8a -- Before Paste

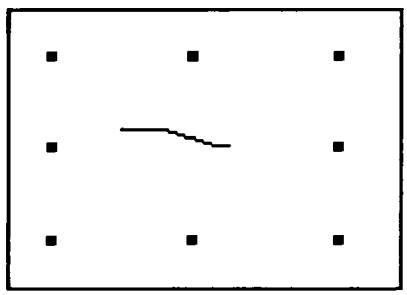


Figure B-8b -- After Paste

Duplicate

Duplicate (⌘D) is simply a **Copy Object** operation immediately followed by a **Paste**. If no object is selected, it has no effect.

Refresh

Refresh is included for application debugging purposes and can be ignored under most circumstances. It re-displays the current page and all its objects, removing stray marks which are not objects.

Clear

Choosing the **Clear** command from the **Edit** menu will blank out the current page and remove all the objects on that page. All that will remain will be the dotted-line border around the white drawing area. [Warning: there is currently no confirmation box to ask you if you really want to do this!] All other pages will remain intact.

The Tools Menu

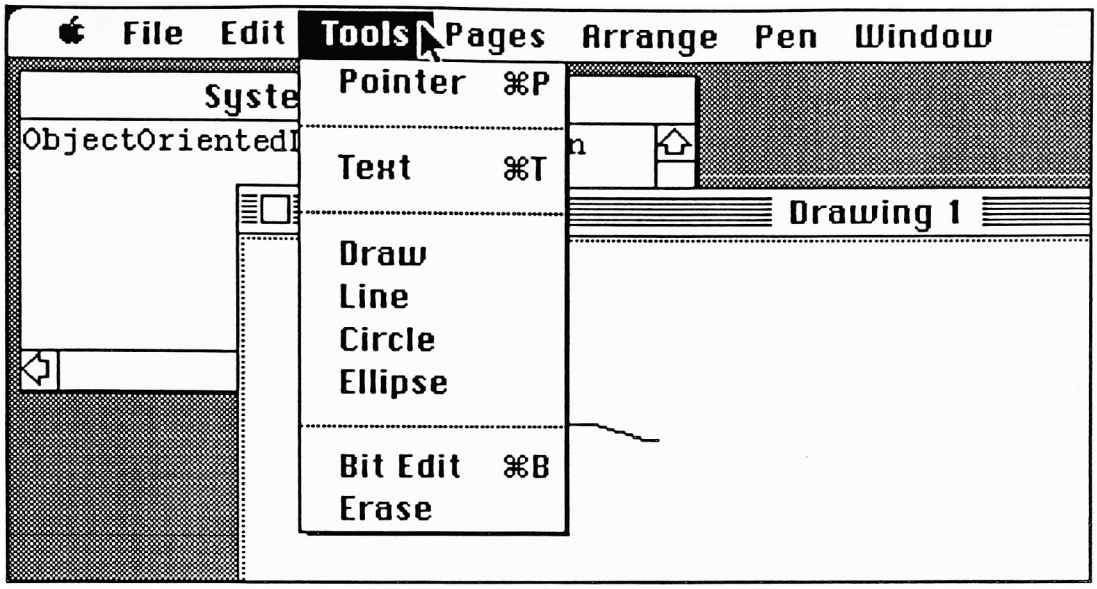


Figure B-9 -- The Tools Menu

Pointer

Moving Objects

Selecting **Pointer** from the **Tools** menu (or typing ⌘**P**) will place you in pointer mode, which is the default mode. The arrow cursor is used to indicate pointer mode. In pointer mode, you may select any object by clicking the mouse button within its bounding rectangle. An object will show that it is selected by displaying its handles -- two to eight small black rectangles surrounding the object (see figure B-10). Clicking on one object to select it

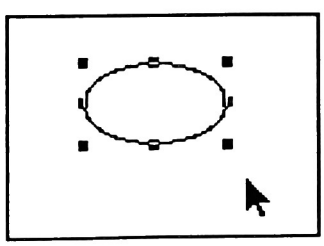


Figure B-10 -- Object with Handles

will deselect any previously selected object (unless the shift key is pressed at the same time -- see below). Clicking on the background area (again, without the shift key pressed) will also deselect any selected object(s).

Once an object is selected, it may be moved by pressing the mouse button within the object's bounding rectangle, dragging the mouse to the desired location, and releasing the mouse button. A dotted-line rectangle of the size of the object's bounding box will follow the mouse's movement, so you can see just where the object will be placed.

The only restriction on dragging is that the object cannot be dragged outside of the drawing's rectangle, or outside of the window (See figure B-11).

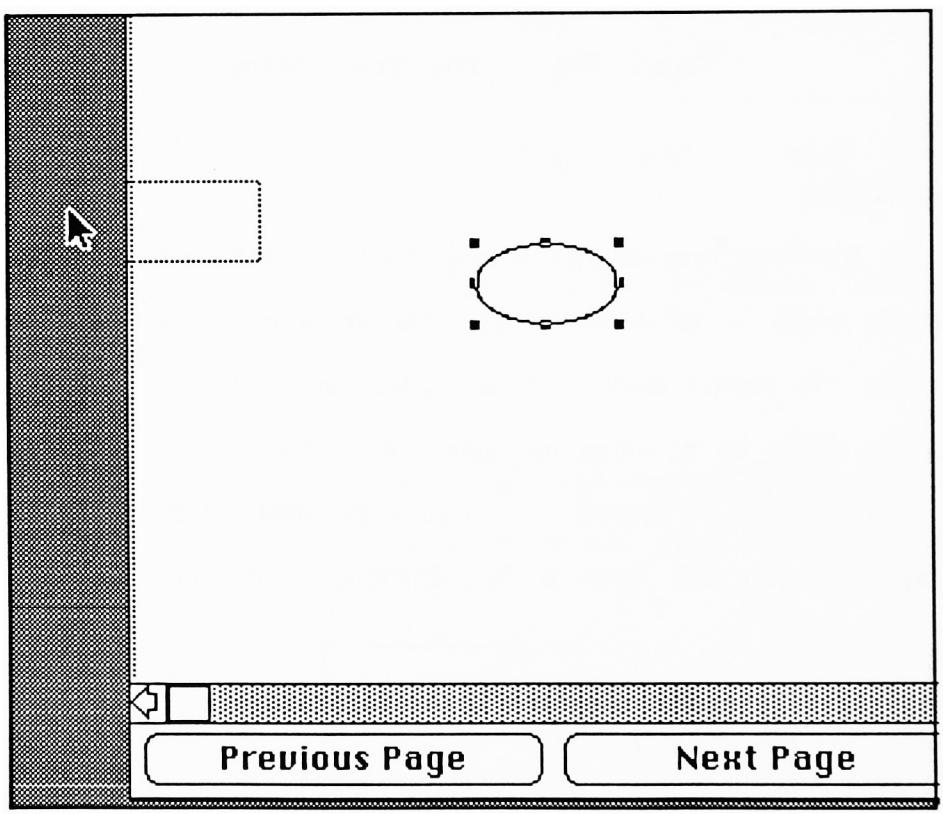


Figure B-11 -- Dragging Restriction

If the window is too limiting, the object can be dragged part way, the window scrolled, and the object dragged the rest of the way. It is possible to shove the

object outside of the drawing's bounding box so that just one or two pixels of the handles are showing, but the object can always be retrieved by grabbing onto those handles once again.

Stretching Objects

The handles have their own purpose. By grabbing onto a handle with the pointer, an object may be resized (stretched or shrunk) in one or two dimensions. You'll know you have grabbed a handle when it disappears. Grabbing a corner handle will allow it to be stretched diagonally (figure B-12a); grabbing a side, top, or bottom handle will allow it to be stretched to that side only (figure B-12b). Lines (which have only two handles: one at each endpoint) may be resized by repositioning their endpoints (figure B-12c). All objects may be resized with their handles except text and point objects.

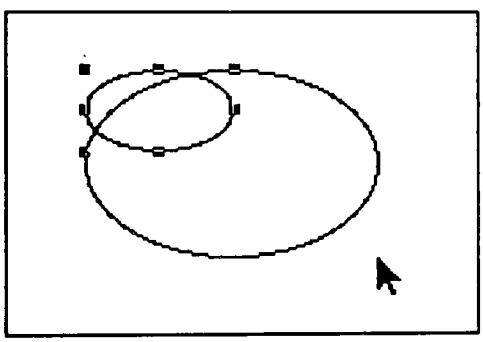


Figure B-12a -- Corner Stretch

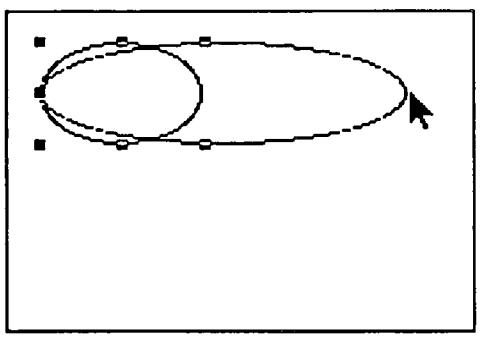


Figure B-12b -- Side Stretch

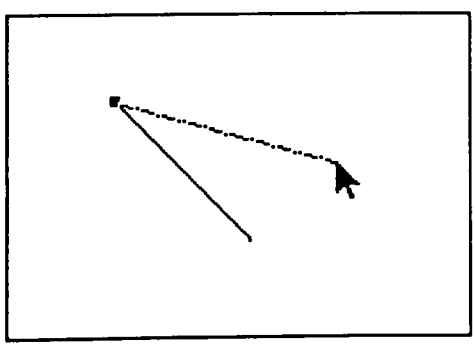


Figure B-12c -- Line Stretch

Multiple Selections

By clicking the mouse at a location outside of any object, then dragging it so that a dotted-line box surrounds one or more objects; all objects within that box will be selected. (See figures B-13, a and b.) If the shift key was not pressed at the time (see below), any object previously selected will turn its handles off.

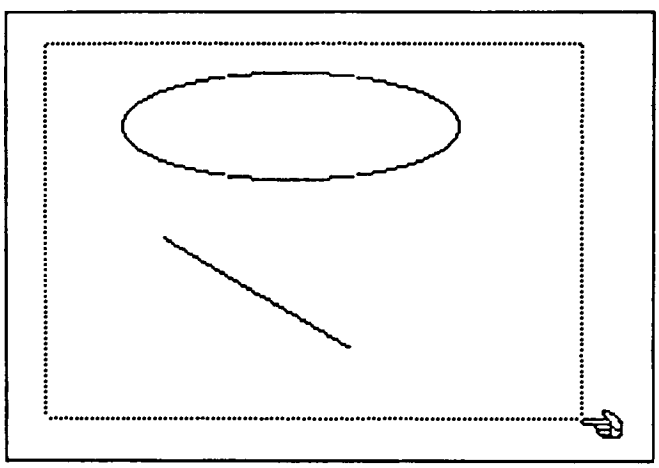


Figure B-13a -- Multi-Select Box

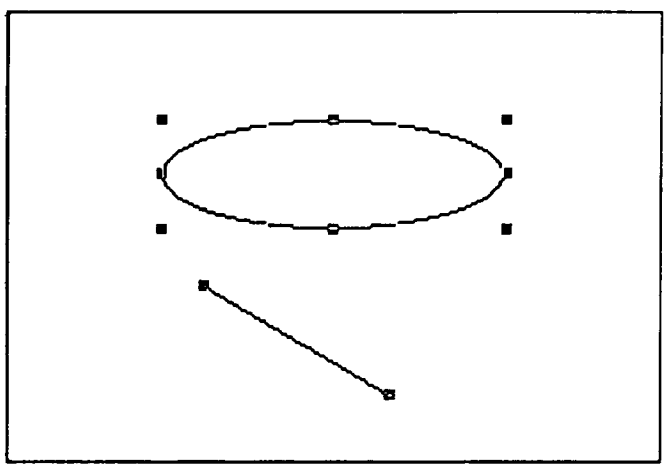


Figure B-13b -- Multiple Selection

A second way of selecting multiple objects is to press the shift key and, while holding the shift key down, click on whatever additional object you want to select (see figure 14). (This eliminates the need for all the selected

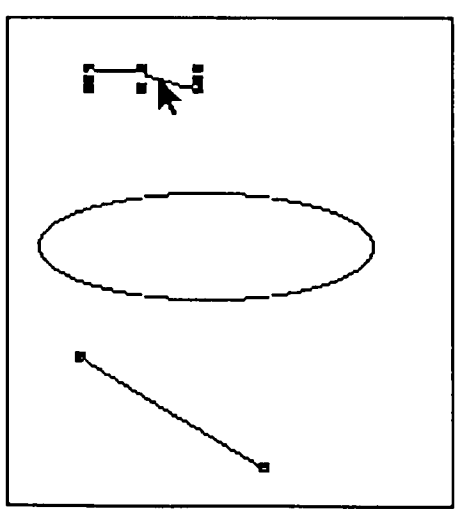


Figure B-14 -- Shift Selection

objects to be adjacent to one another in order to select them together.) If the object was not selected, it will display its handles along with all the other objects which were already selected. On the other hand, if it was already selected when you clicked on it with the shift key down, it will be deselected (its handles will turn off).

In a variation that is a combination of the above two methods, you may surround more than one object with a selection box while the shift key is pressed. This will turn on the handles of any objects in the box that were not already selected, and turn off the handles of any objects that were selected. It will not effect the selection of objects outside the selection box. (See figures B-15a and B-15b.)

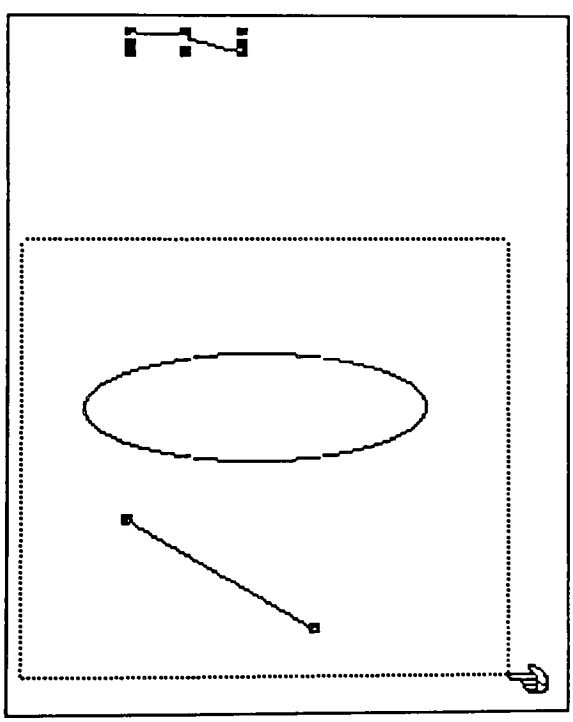


Figure B-15a -- Multi-Shift Select

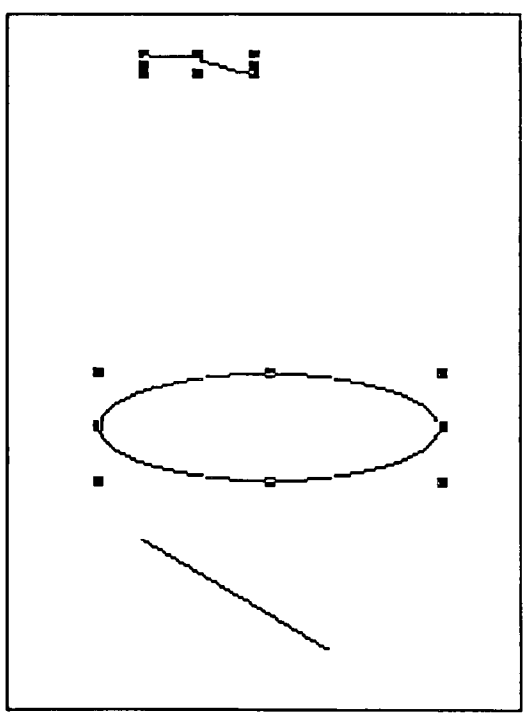


Figure B-15b -- After Multi-Shift

Text

Textual objects may be created by choosing **Text** from the **Tools** menu or by typing ⌘T. An I-bar cursor will appear to let you know you are in text mode.

To create a new textual object, position the mouse where the text should begin. A mouse click and release will create a small rectangle capable of holding one character (see figure B-16a). As characters are entered from the keyboard, this area will grow to accommodate the characters (figure B-16b). It will grow to the right as characters are entered on a single line; and will grow downward each time the return key is typed.

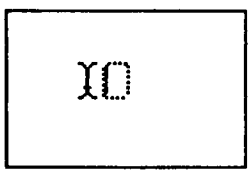


Figure B-16a -- Text Mode

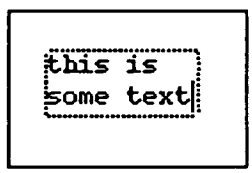


Figure B-16b -- After Typing

Existing text may be manipulated in **Text** mode using standard word processing techniques. When an existing text object is selected (by clicking on it) in Text mode, it will have the dotted-line border around it. The I-bar may be placed anywhere inside the text object, then characters may be inserted, deleted, or highlighted and changed (by typing over them). Highlighted text within a text object appears in reverse-video (see figure B-17). Highlighted text may also be copied, cut, and pasted (described below).

Figure B-17 -- Selected Text

Pasting Text

If text is on the clipboard, it may be pasted into the drawing. How it is pasted depends, however, on the current state of the drawing.

If you are in any mode other than text mode, the pasted text object will behave like other objects: it will be surrounded by eight handles, and you will be put into pointer mode. (See figure B-18.) If the object was cut or copied

from this or another drawing, it will have retained its font and point size information, and will be pasted into the drawing with its original font and point size.

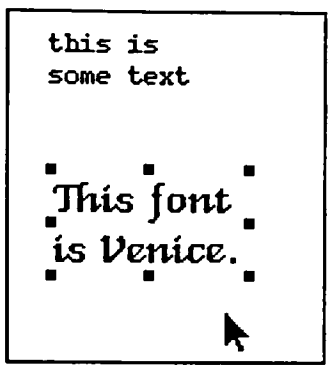


Figure B-18 -- Pasted Text, Pointer mode

If you are in text mode, but no text object is selected (no text object is surrounded by dotted lines), the object will be pasted in as above, retaining its font and point size, and you will be placed into pointer mode.

If you are in text mode, and click (or click and drag) the mouse to create a new (empty) text object, then paste, the object will be pasted into this empty text object, and retain its original font and point size, and you'll stay in text mode. (Because the new text object was empty, it had not yet taken on the characteristics of the current text font; see below.) The pasted text will be highlighted (reverse video).

If you are in text mode, and an existing (non-empty) text object is selected (has a dotted line surrounding it), the text from the clipboard will be pasted a) wherever the I-bar cursor is within that selected text object, or b) over whatever text is highlighted, replacing the highlighted text. The pasted text will then be highlighted. (See figure B-19.)

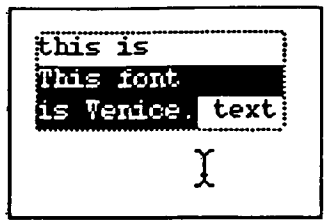


Figure B-19 -- Pasted Text, Text mode

Changing a Text-Object's Font

No matter what the font and point size of the text on the clipboard, once it is pasted into an existing, non-empty text object, it assumes the font and point size of that existing text object. This is true of any existing non-empty text object. That means that any text object may have its font changed to the current font by doing the following.

1) Cut or copy the text object.

2) Change the font to the one desired, if it isn't already the current text font. (See **Change Text Font...** under **Window Menu** below.)

3) Create a new text object in the current font, and type some letter (even a space) into it.

4) Paste the text into this new, one-character text object.

5) Remove the one character.

Getting Text to the Clipboard

Text may get to the clipboard in any of several ways:

1) It may have been cut or copied from some other (perhaps word processing) application. In this case, it will have no formatting information, and will be pasted into the drawing in whatever font is the current text font. (If it was cut from one paragraph of an application which word-wraps at the right margin, it will be one long, unwrapped line.)

2) It may have been cut or copied from the drawing while the drawing was in pointer mode. This is done the same way that any other selected object (an

object with its handles visible) is cut or copied. (See above.) In this case, it will have been an entire text object, and its font information will have been cut or copied to the clipboard with it. [Note: this last statement means that if you then attempt to paste this text object into another (perhaps word processing) application, you will get, along with it, font and point size formatting information that the other application will not understand. It is trivial to edit this out, but you should be aware that it is there.]

3) It may have been cut or copied from the drawing while the drawing was in text mode. In this case, it could have been all or part of an existing text object. To cut or copy part of an existing text object, go to text mode and highlight the text to be cut or copied. (Click the mouse button at one end, drag the mouse to point to the other end, and release the mouse button.) Then select either **Cut** or **Copy Object** from the **Edit** menu. Cut text will be removed from the existing text object. Both operations will put the text and its formatting information on the clipboard.

Duplicate, since it is a copy operation immediately followed by a paste operation, has no visible effect in text mode. What was highlighted gets put on the clipboard, and what is on the clipboard replaces what is highlighted, leaving the result highlighted. (Its invisible effect is to change the clipboard.)

Cut, **Copy Object** and **Duplicate** all make the terminal beep if you are in text mode, a text object is selected, and no text is highlighted.

Something you should not do is use **Copy Rectangle** to copy text. While it will put whatever rectangular area you select on the clipboard (including rectangular areas that have text in them), it puts the area there as a bitmap, not as text. Therefore, when you paste it back into your drawing, you will not be able to edit the characters as text, even though what you see looks like text.

Draw

Draw allows you to do free-hand drawing. When the mouse button is up, no marks will show up on the drawing. When the mouse button is pressed, a line the size of the pen size (see **Change Pen Size...** below) will begin tracking the mouse (figure B-29a). When the mouse button is again released, one freeform drawing is completed, and its handles will appear (figure B-20b). Therefore, every mouseDown-mouseMove-mouseUp sequence constitutes one freeform object. The pencil cursor lets you know you are in draw mode.

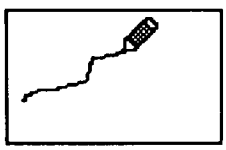


Figure B-20a -- Draw, mouse down

Figure B-20b -- Mouse up

Line

Lines start where the mouse button is depressed, and end where the mouse button is released. These lines may be drawn at any angle with no restrictions. When the mouse button is released, the line's two handles appear at its endpoints. (See figure B-21.) The star cursor lets you know you are in line mode.

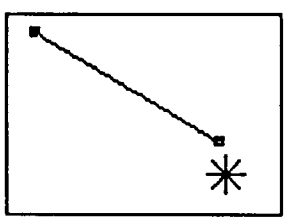


Figure B-21 -- Line Mode

Circle

The **Circle** tool allows you to create circles drawn from the center outward. A cursor in the shape of a small circle lets you know you are in circle mode, and allows for a precise placement of the center of the circle. This is done

with an initial mouse click. As the mouse is dragged, with the button down, in any direction from the center, the cursor changes to indicate where the mouse is, relative to the center (see figure B-22a). (The cursor shape becomes a partial curve, always curving toward the center of the circle.) When the mouse button is released, the circle is complete, and its handles are displayed (figure B-22b).

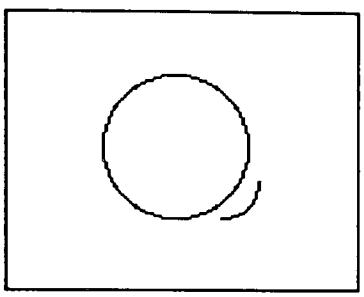


Figure B-22a -- Circle, mouse down

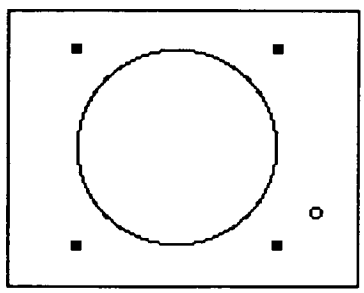


Figure B-22b -- mouse up

Ellipse

The **Ellipse** tool behaves similarly to the circle, but draws either horizontal or vertical ellipses. (See figure B-23.) Its cursor is a small ellipse. Notice that circles have four handles and ellipses have eight. This is partially to distinguish circles from ellipses which are (temporarily, perhaps) round. It is also because ellipses may be stretched horizontally and vertically, and circles can only be stretched diagonally.

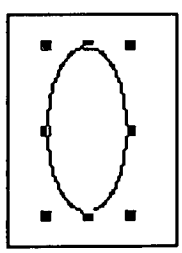


Figure B-23 -- Ellipse

Bit Edit

Any selected object may be bit-edited by selecting **Bit Edit** from the **Tools** menu or by typing ⌘**B**. Another window will open, showing the object at eight times its normal magnification (see figure B-24) and the object will be centered in this window. In the lower left-hand corner is an inset pane

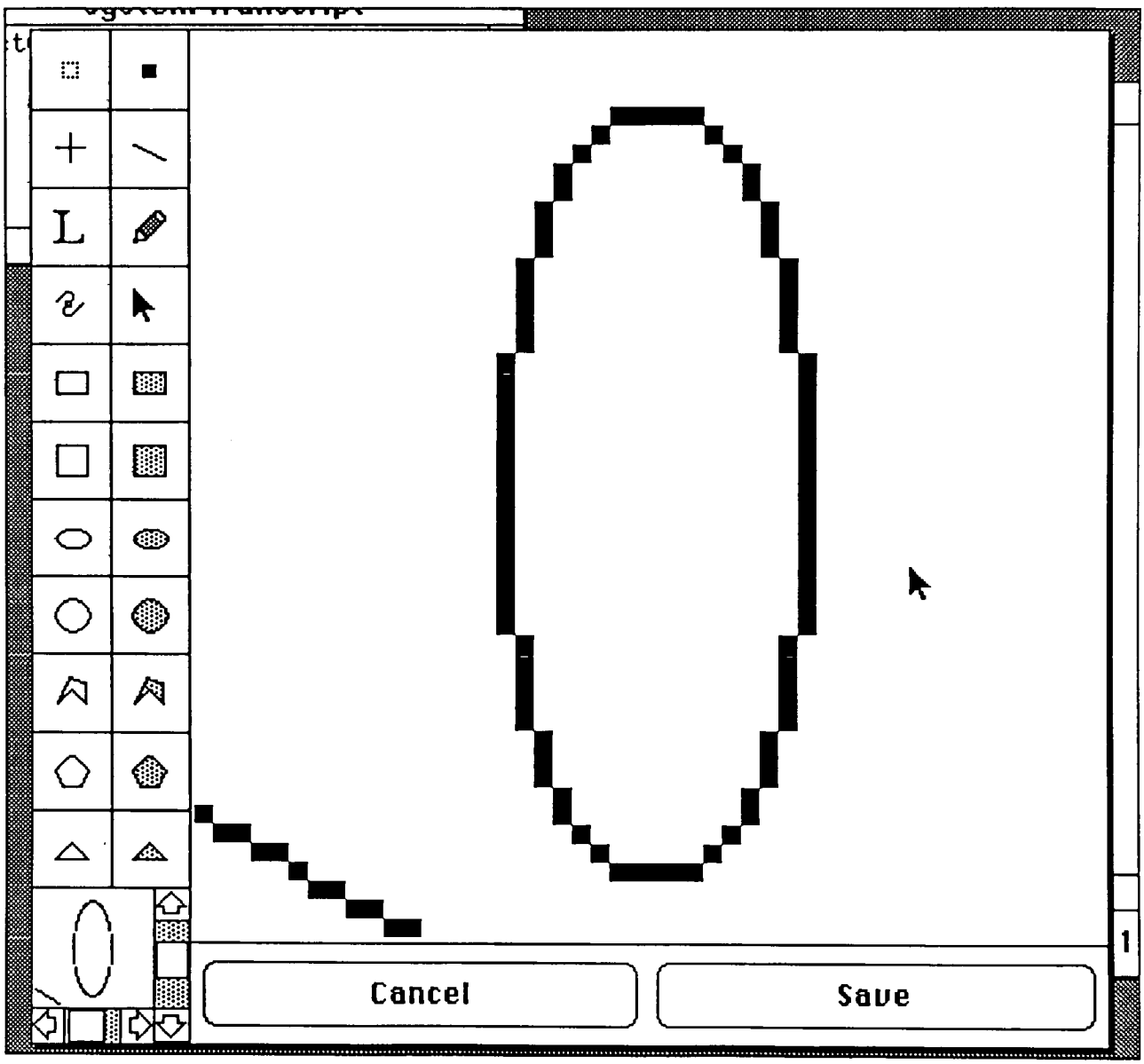


Figure B-24 -- The Bit Editor

showing the same portion of the drawing at its normal magnification. This inset pane has scroll bars: when it is scrolled, the bit pane will also scroll.

The Palette

The bit editing window also has a tool palette on the left side: a panel of icons for the different tools that are available in the bit editor. Clicking the mouse over one of these icons will highlight it and put you in the mode of that tool. (The cursor will change to reflect this mode.)

White Points

The tool at the upper left is the white-point tool. When this tool is active, white points will be placed on the drawing wherever the mouse is clicked. (If the mouse is held down and moved, white points will follow it.) This is useful for covering up with white the edges of a black object (see figure B-25). White points are visible on a white background because they are displayed with gray borders.

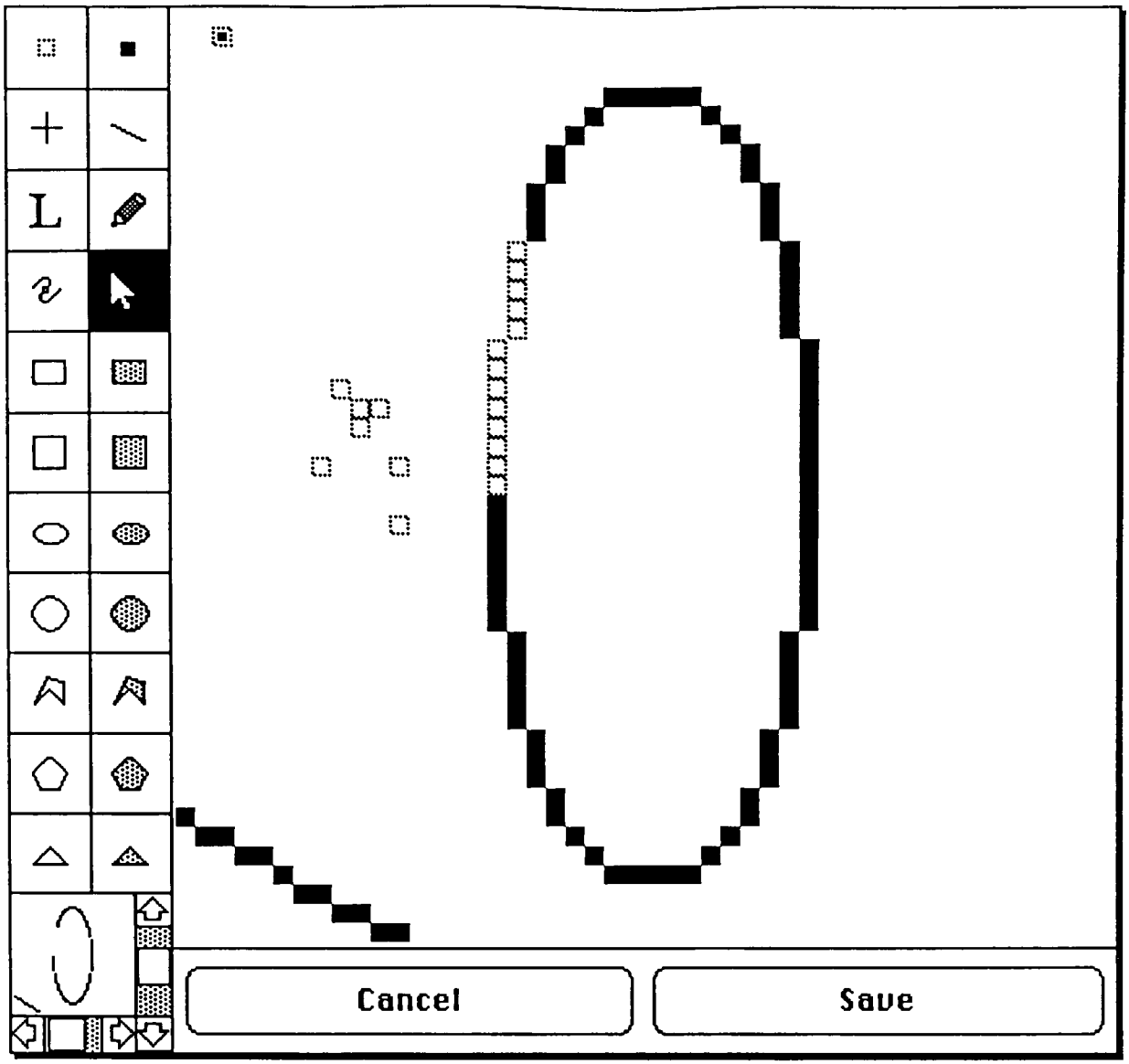


Figure B-25 -- White Points

With the pointer tool, points (both white and black, see the next section) can be selected and moved, cut, copied and pasted. Many points may be selected at a time and moved together. A point will show that it is selected by displaying one handle in the middle of the point. This handle will reverse the inside of the point, so a selected white point will have a small black square in it, and a selected black point will have a small white square in it. (For example, both the upper-left-most point in figure B-25 and the lower-right-most point in figure B-26 are selected.) Points are not stretchable.

Black Points

To the right of the white point on the palette is the black point. Black points are created the same way white points are: they will be placed wherever the cursor is clicked. Figure B-26 has several black points to the right of the ellipse. Notice that these points are shown slightly smaller than the points which make up the edge of the ellipse. (This is an effect in the bit editor only. Each point is one-pixel square in actual size.) They are also separate from one another. Both features emphasize that they are points.

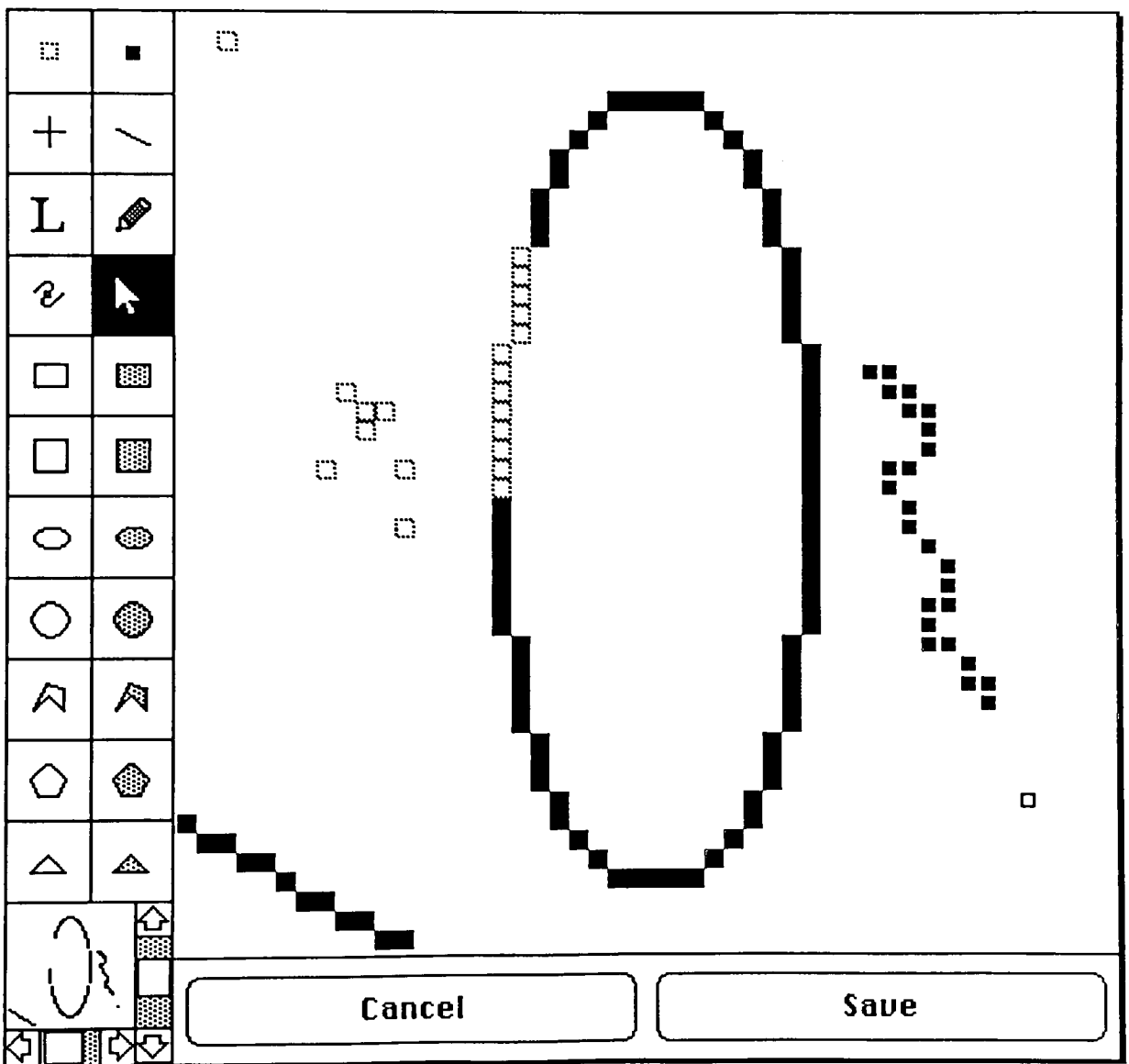


Figure B-26 -- Black Points

Line

Below the black point is the line tool. Lines may be placed on the drawing in the bit editor just as they are outside the bit editor: click the mouse at the starting location, and hold it down until you've reached the ending location. Figure B-27 shows a new line in the Bit Editor. Notice that because it is at a higher magnification, it looks like "stair-steps" in the bit editor.

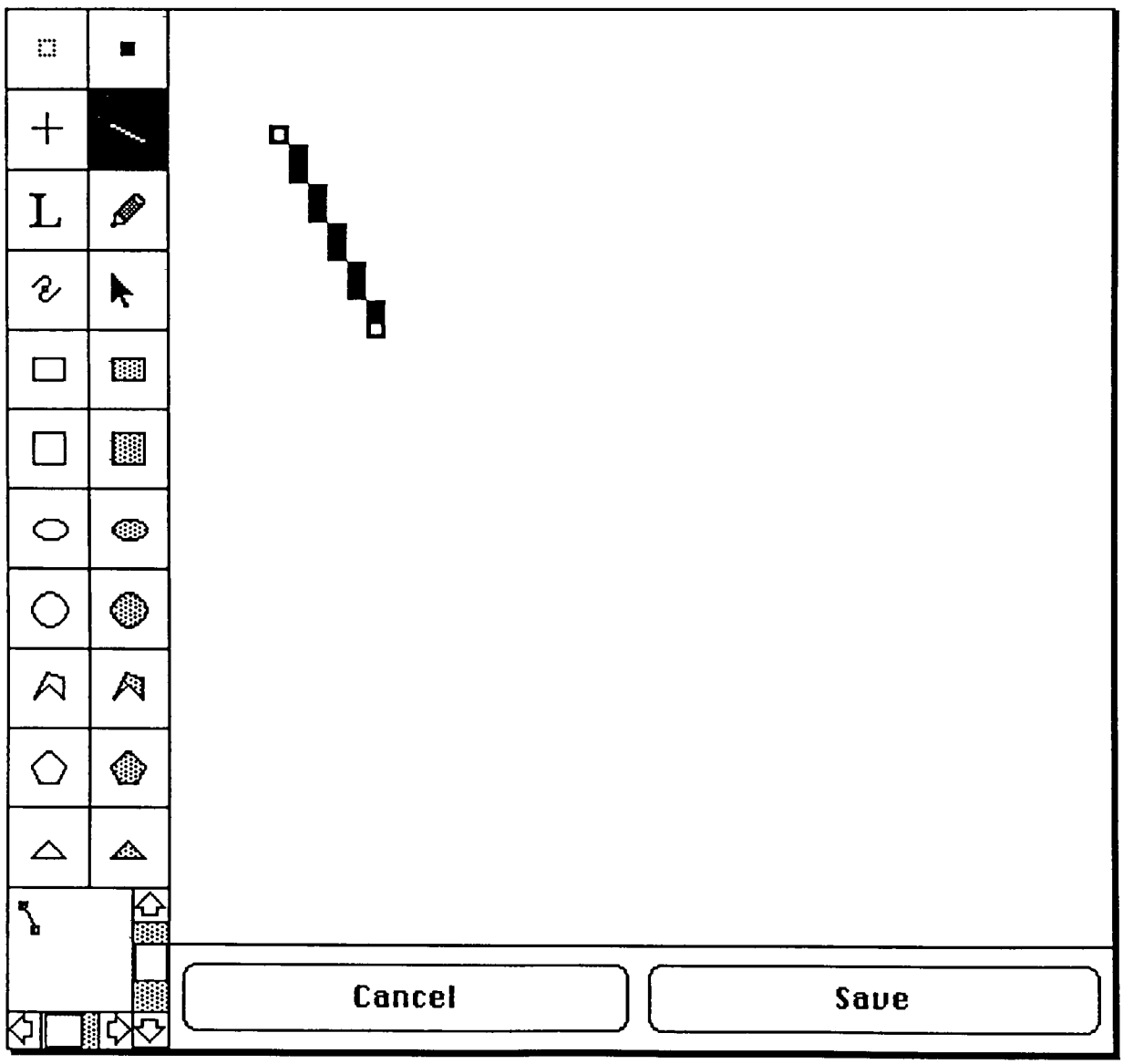


Figure B-27 -- Line in the Bit Editor

Pencil

Below the line tool is the pencil tool. This behaves the same way as the draw tool outside of the bit editor. (See **Draw** above.) Figure B-28 shows a new freeform object to the right of the black points. Notice that the bits of the freeform are shown larger than those generated with the point tool. (Again, this is just for visual distinction in the bit editor. The points are actually the same size in the drawing.)

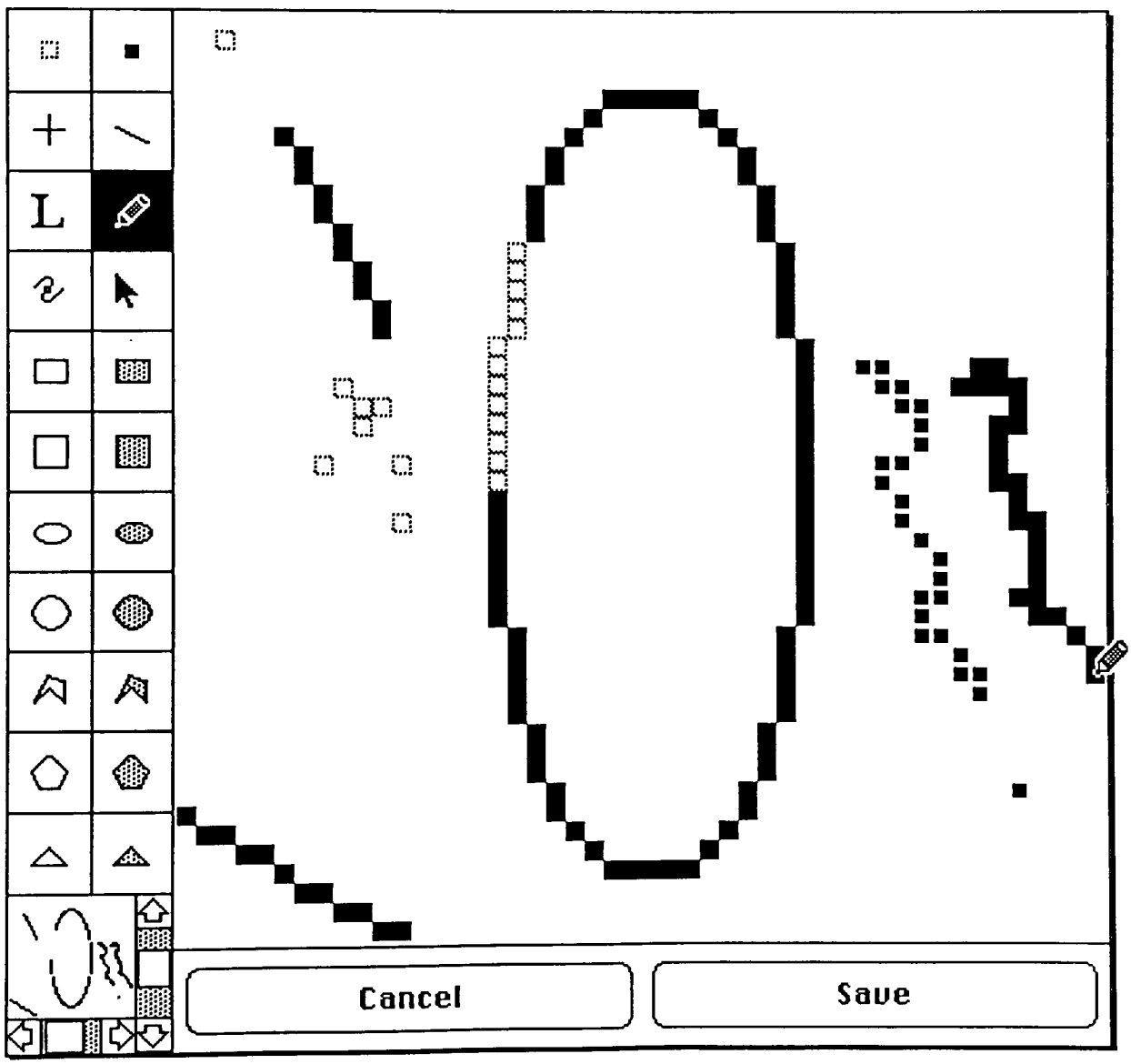


Figure B-28 -- Pencil Tool in the Bit Editor

Arrow

Below the pencil tool, in the shape of the arrow, is the pointer tool. This works the same as the pointer tool in the drawing (see **Pointer** above): objects may be selected, multiply selected, stretched, grouped, etc.

Others

None of the other tools in the palette are available at this time. You may click on one, but nothing will happen. They are there for future expansion.

Leaving the Bit Editor

You may leave the bit editor in one of two ways: by saving your changes, or by throwing them away. If you click the **Save** button at the bottom of the window, the bit editor will close and your changes will be reflected in the original drawing. If you click **Cancel**, the bit editor will close and your original drawing will be unchanged.

The Bit Editor Menus

The menus are all the same as those in the drawing window; however, two of the commands have added functions in the bit editor and deserve further mention.

Ungrouping Bitmaps

In the bit editor (*only*) you may ungroup into their points bit-mapped objects: those which were created by pasting in a rectangularly-copied area; or those created with the pencil or draw tool. Simply select any bit-mapped object and choose **Ungroup** from the **Arrange** menu. After a short pause (the duration of which depends on the size of the object), you will see all the points which make up the bitmap become smaller and separate, and they will all be selected. Figure B-29 shows the freeform object from figure B-28 ungrouped.

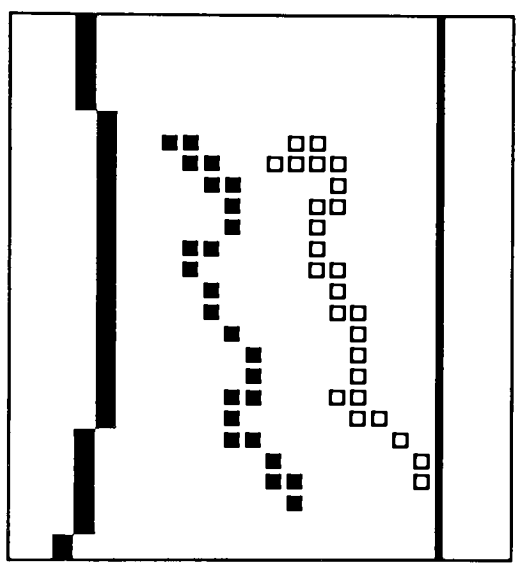


Figure B-29 -- Freeform Object Ungrouped

Grouping Points

Point objects may not exist outside of the bit editor. You may, however, group them into a bit-mapped object before you leave the bit editor. Simply select the points you want grouped together and choose **Group** from the **Arrange** menu. A bit-mapped object with its larger-looking bits will replace the points. Figure B-30 shows the black points from figure B-28 grouped.

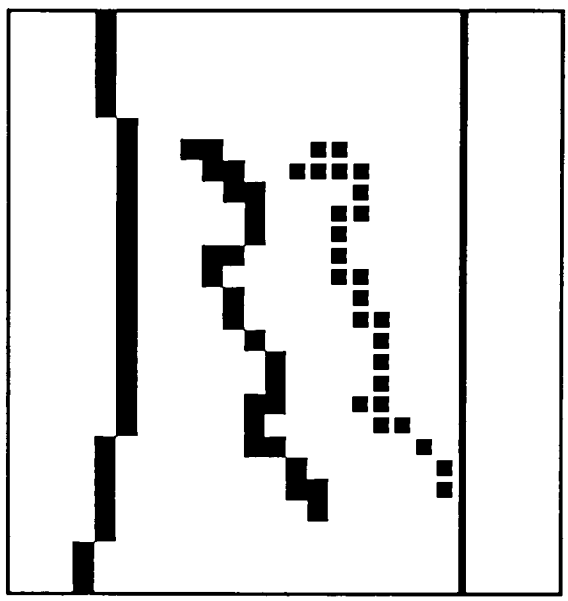


Figure B-30 -- Bits grouped

Restriction on Point Groups

You may not group together both black and white points in the same bit-mapped object. If you try, a dialog box will appear alerting you that you have attempted this (see figure B-31). It will also give you four choices: 1) you may have it turn all the points black and group them together; 2) you may have it turn all the points white and group them together; 3) you may have it make two groups (one black and the other white); 4) you may cancel the grouping operation.

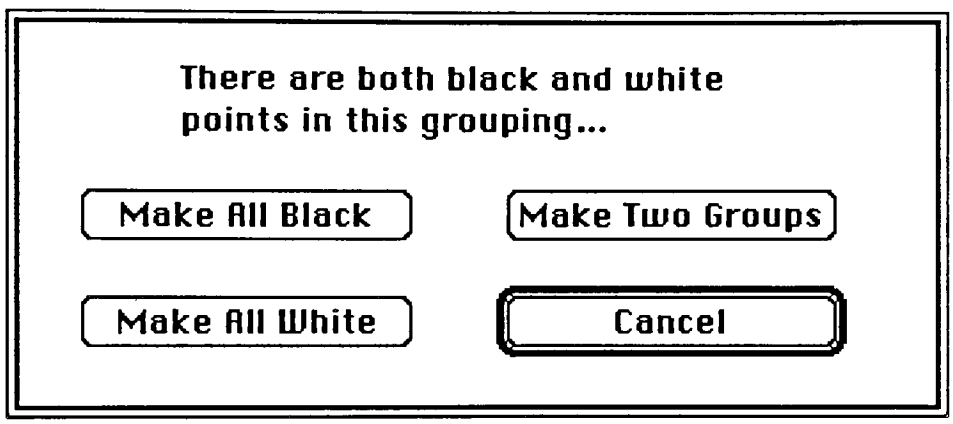


Figure B-31 -- Black/White Dialog

Grouping on Save

Finally, because points may not exist outside of the bit-editor, when you try to leave the bit editor by clicking the **Save** button, it will check for ungrouped points. If it finds any, a dialog box will appear (figure B-32) asking you if you would like it to group those points together (press **Group All**) or cancel the closing and saving of the bit editor (press **Don't Close**). If you press **Group All**, the bit editor will try to group all the points into one point object. If it finds both black and white points, it will give you a choice whether to make them all black, all white, or make two groups, as above.

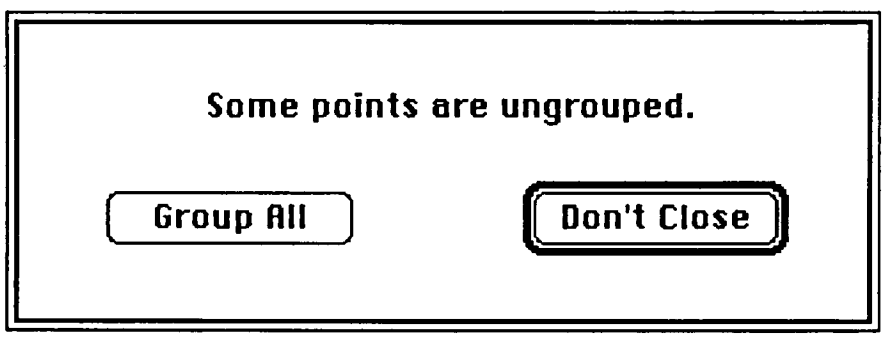


Figure B-32 -- Auto-Group Dialog

Erase

The **Erase** command allows you to remove large parts of an object at a time, and works only on freeform and pasted objects. An eraser-shaped cursor lets you know you are in erasure mode. An object may be selected before going into erasure mode, or you can select an object while in erasure mode by clicking on the object. Whenever the eraser cursor is inside an object and the mouse button is down, the object will become white where the cursor is. (The white area will be the shape of the eraser.) For more precise erasure than the eraser tool gives, use **Bit Edit**.

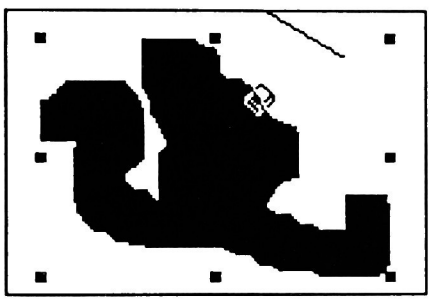


Figure B-33a -- Eraser Cursor, Mouse Up

Figure B-33b -- Mouse Down

The Pages Menu

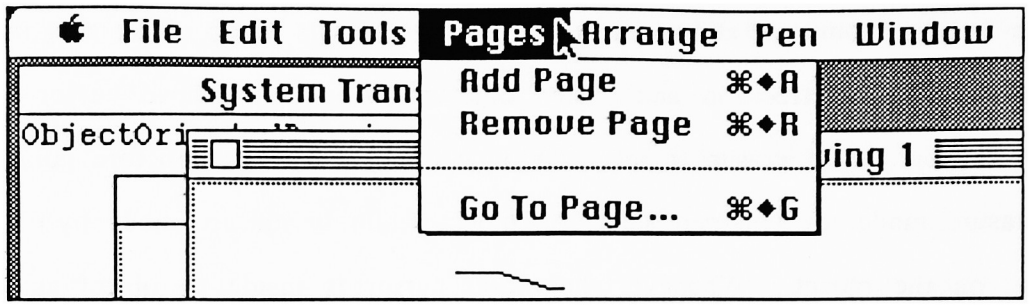


Figure B-34 -- The Pages Menu

Add Page

A new drawing page may be added by either selecting **Add Page** from the **Pages** menu or typing ⌘-**shift**-**A**. New pages will always be added at the end of the document, and you will immediately go to that new page. If you are on page one of two pages, for example, and invoke **Add Page**, you will go to a new page 3. The window always shows you in its lower right-hand corner which page you are on and how many pages there are (e.g. "Page 1 of 3").

If you are already on the last page of the document, another way to add a page is to press the **Next Page** button on the bottom of the window. For example, if you are on page 3 of 3 and press the **Next Page** button, you will be on page 4 of 4.

Remove Page

The current page may be removed by selecting **Remove Page** from the **Pages** menu or typing ⌘-**shift**-**R**. If you are on any page other than the first page, it will remove that page and take you to the previous page. If you are on the first page, it will remove it and take you to the next page. If there is only one page, however, this option is disabled; the drawing must always have at least one page. [Warning: there is no dialog box which asks if you really want to remove a page, even if you have made significant changes to it.]

Go To Page...

You may flip through the pages by clicking on the **Previous Page** and **Next Page** buttons at the bottom of the window. If there are many pages, however, you may prefer to go directly to the page of choice. Select **Go To Page...** from the **Pages** menu or typing **⌘-shift-G**. A dialog box will appear asking you which page you'd like to see (figure B-35). You may type in the page number and press the **Accept** button, in which case it will take you to that page; or press **Cancel**. If you type in a page number which is larger than the total number of pages, it will take you to the last page of the document. If the number is less than 1, it will take you to the first page.

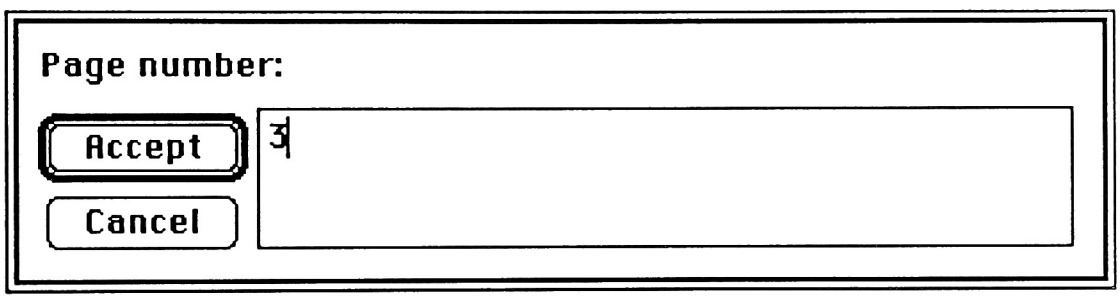


Figure B-35 -- Go To Page Dialog

The Arrange Menu

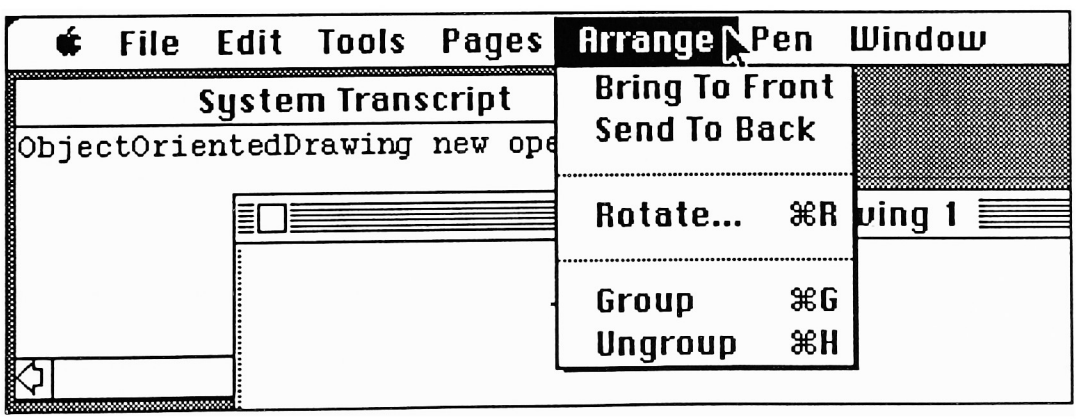


Figure B-36 -- The Arrange Menu

Bring To Front and Send To Back

When an object is either pasted into the drawing or created as a new object, it will be, by default, the front-most object. In other words, if an older object is moved into the bounding rectangle of a newer object, the newer object will appear to be in front of the older. The newer object will also be the first of the two selected when the mouse is clicked inside the bounding box of both objects.

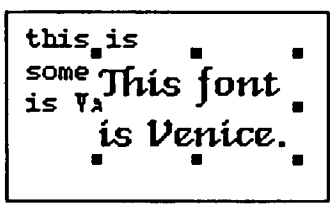


Figure B-37a -- Before Send To Back

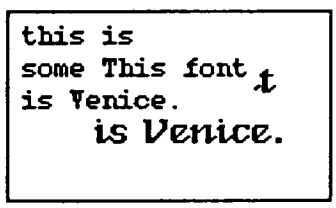


Figure B-37b -- After Send To Back

While in pointer mode, this front-to-back orientation can be changed with the **Arrange** menu commands **Bring To Front** and **Send To Back**. An object should be selected first; then the desired command chosen from the menu (see figures B-37, a and b). The objects will be displayed in their new front-to-back orientation, and will stay in this orientation until another similar command changes it. (New objects will still become front-most as they are created, however.) If no object is selected when one of these commands is executed, no change will occur.

Rotate...

Freeform objects (objects created with the **Draw** tool) and objects created by copying a rectangle and pasting it back into the drawing (in other words, all bit-mapped objects) may be rotated. Select **Rotate...** from the **Arrange** menu. A

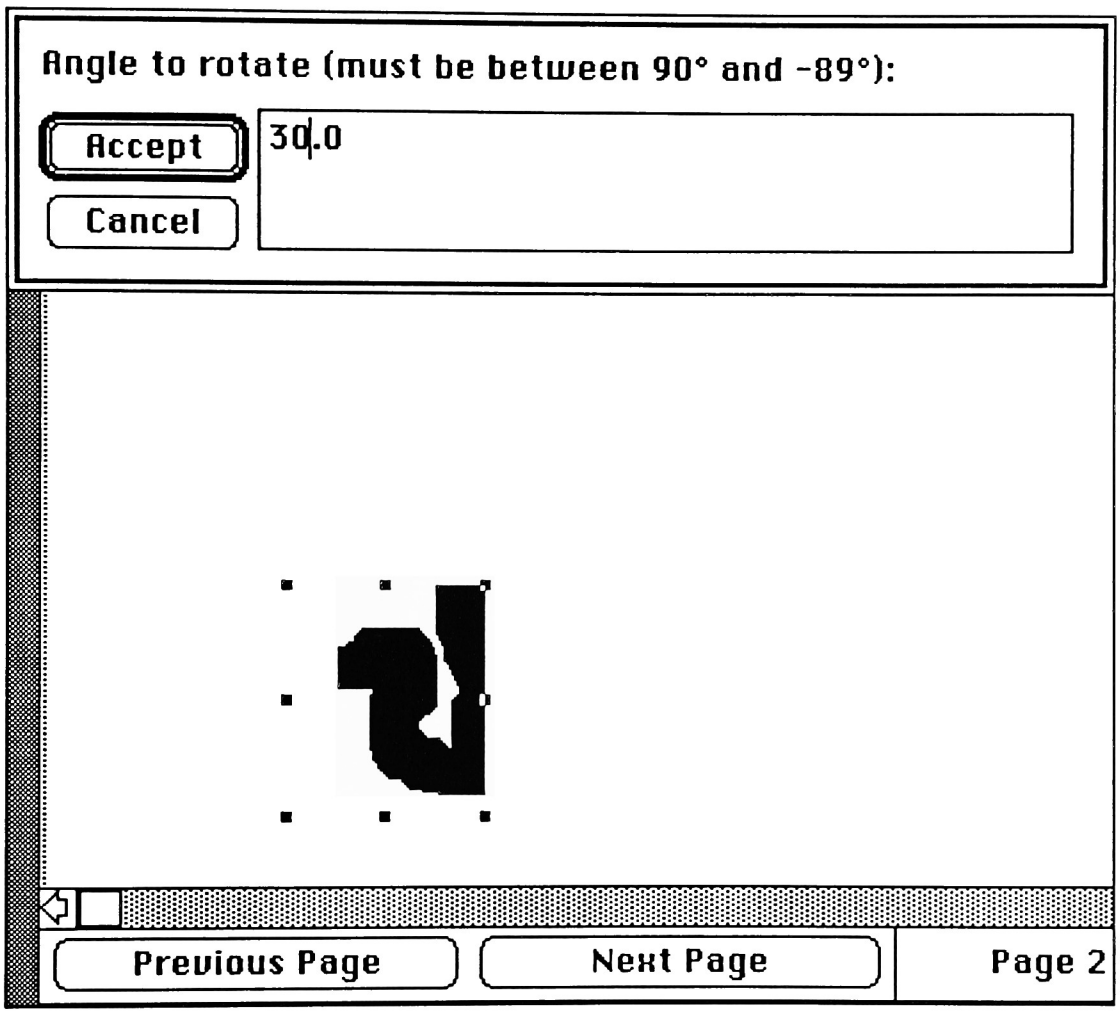


Figure B-38 -- Rotate Dialog

dialog box will appear in which you may type the angle of rotation to a single degree of precision (see figure B-38). This angle must be between positive 90° and negative 89°, inclusive. If the angle is not in this range, the dialog box will continue to prompt you for a valid angle. If you decide not to rotate the object, you may press the **Cancel** button on the dialog.

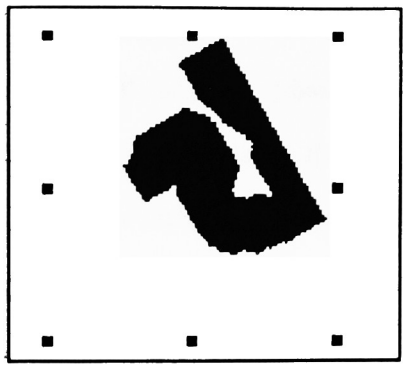


Figure B-39 -- Object Rotated 30°

Note that the angle of rotation is the angle from zero (the object with no rotation). It is not the angle of rotation relative to the current angle of rotation (i.e., rotating an object to 45° twice will not put it at 90°).

Group

When more than one object is selected, you may **Group** them together to form a single object. The objects need not be touching or even near each other. After the grouping operation, the resulting composite object will be displaying its handles. (See figures B-40, a and b.) If only one object is selected, **Group** will have no effect.

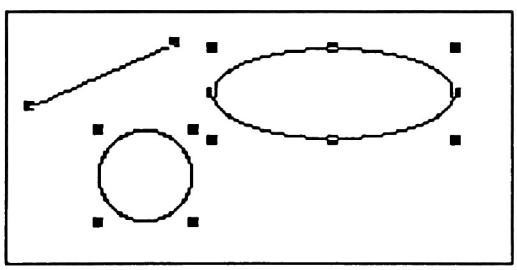


Figure 40a -- Before Grouping

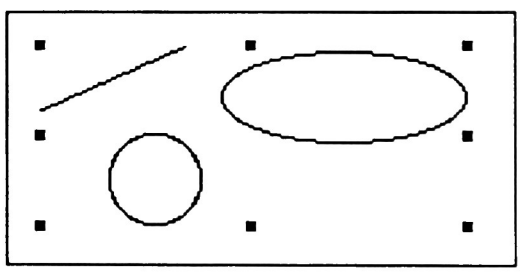


Figure 40b -- After Grouping

Ungroup

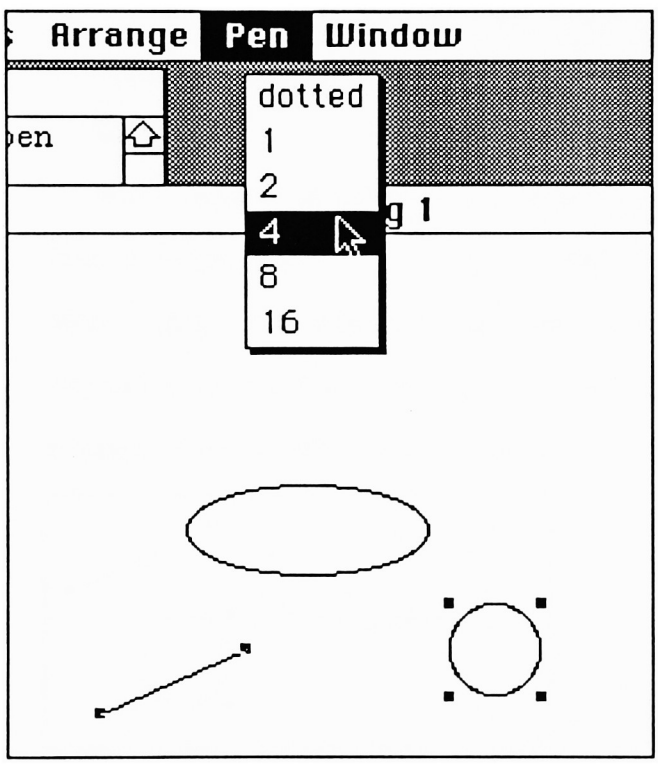
Any object which has previously been grouped (i.e., any composite object) may be ungrouped. After the ungrouping operation, the objects which made up the composite object will all be displaying their handles. [If one of the objects which made up the composite was itself a composite, it will maintain its grouping, i.e., it will not be ungrouped into its component parts until another **Ungroup** operation is performed.] More than one composite may be ungrouped at a time: if more than one is selected when the **Ungroup** command in invoked, they will all be ungrouped. If an object selected when the **Ungroup** command is invoked is not a composite, the command will have no effect on that object.

The Pen Menu

Figure B-41 -- The Pen Menu

Change Pen Size...

To change the drawing pen size, select **Change Pen Size...** from the **Pen** menu. Several fixed pen sizes are available, including a dotted line (see figure B-42a.) Changing the pen size when any lines, circles, or ellipses are selected



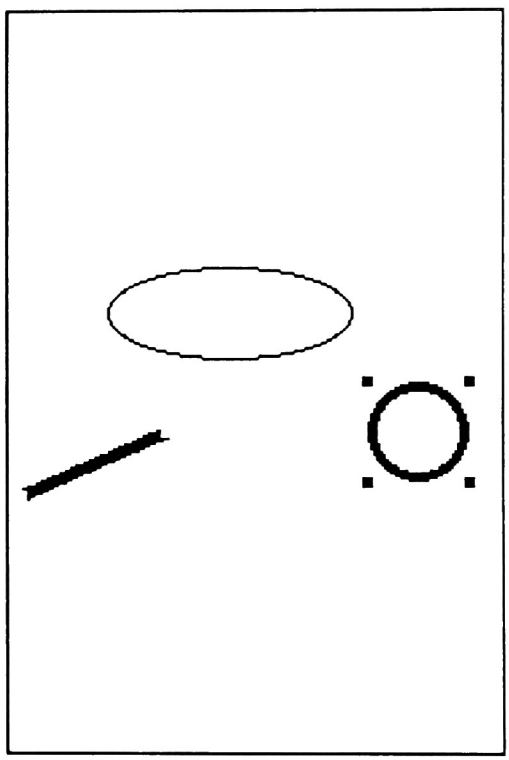


Figure B-42a -- Pen Size Choices **Figure B-42b -- After Changing Size**

will change those objects' line thicknesses and all subsequent lines, circles, ellipses, and free form drawings will be drawn in the new size. If no object is selected when **Change Pen Size...** is chosen, only future objects will be affected.

The Window Menu

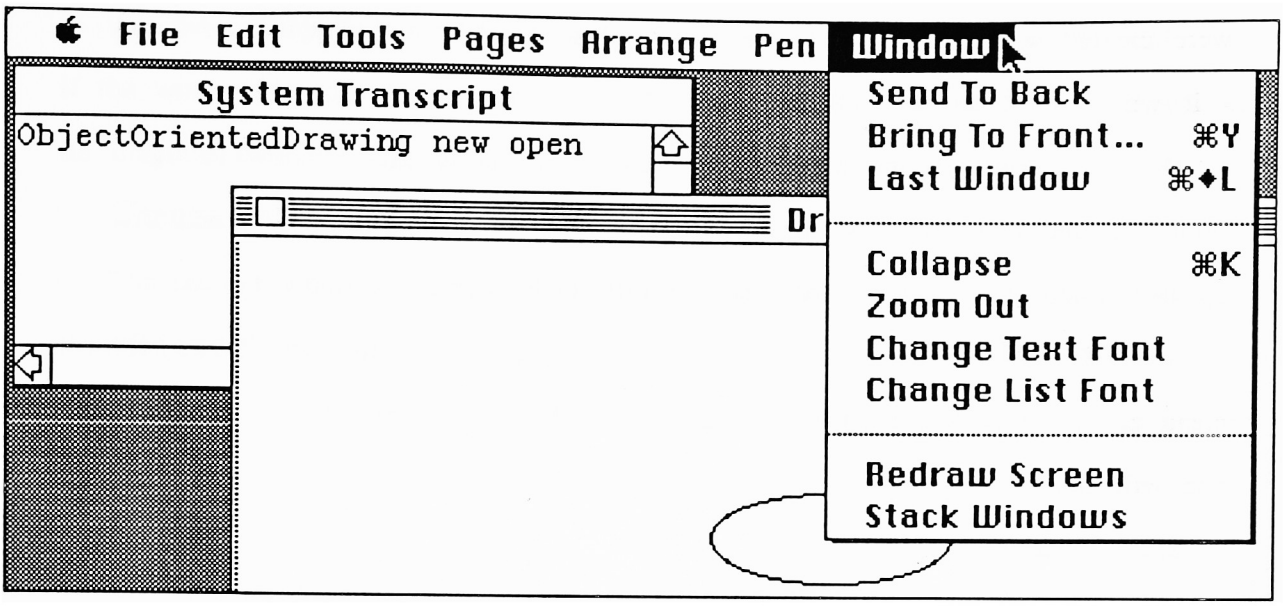


Figure B-43 -- The Window Menu

Send To Back

Windows themselves may also be rearranged. Select **Send To Back** from the **Window** menu, and the current window will be placed behind all the other windows.

Bring To Front

Any window may be brought to the front with the **Bring To Front** command (or ⌘Y). A short, wide window will appear at the bottom of the screen showing the titles of all the windows (see figure B-44). Whichever window you had at

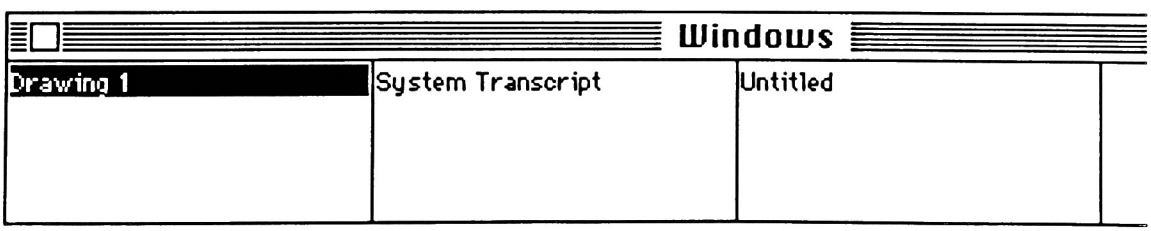


Figure B-44 -- The Windows window

the front when you invoked **Bring To Front** will be highlighted. If there are two or more windows with the same name, they will be numbered. (The

numbers may appear somewhat arbitrary. They are loosely in the order they were created, although if you collapse and then expand a window -- see below - it will have a higher number.)

Double-clicking on the name of a window will bring that window to the front. If you wish, you may leave this **Windows** window open; it will be updated when you click on it again to reflect the current window names. Alternatively, if it is already open, selecting **Bring To Front** from the **Windows** menu again (or typing ⌘Y) will bring the **Windows** window to the front (a new one will not be created).

Last Window

Whichever window you had at the front prior to the previous window may be re-selected by choosing **Last Window** from the **Windows** menu or by typing ⌘-**shift**-**L**. (If the last window selected was the **Windows** window, it will not be brought to front, since it may be brought to front by typing ⌘Y. Instead, the window viewed prior to the **Windows** window will be selected.)

Collapse/Expand

A window may be reduced to only its label area (see figure B-45) by choosing **Collapse** or typing ⌘K. If a window is collapsed, the command changes to **Expand**; it may be returned to its original size by choosing **Expand**, or by typing ⌘K again.

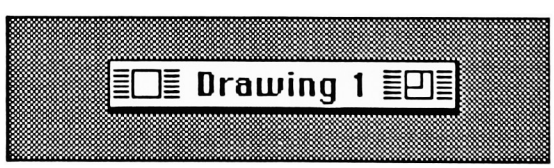


Figure B-45 -- A Collapsed Window

Zoom In/Out

A window may be made to fill up the entire screen by choosing **Zoom Out**. If the window is zoomed, the menu item becomes **Zoom In**; it may be returned to its original size by choosing this command.

Change Text Font

The current font may be changed using the **Change Text Font** option under the **Window** menu. A dialog box will appear which has two pop-up menus on it: one for the text font and one for the font size. Simply press and hold down the mouse button over the font name or point size, and the list will pop up. (See figures B-46, a and b.)

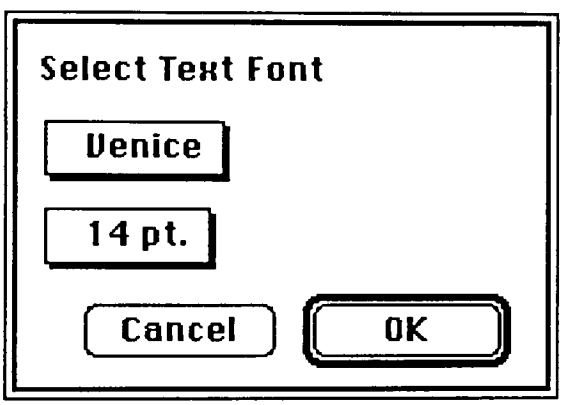


Figure B-46a -- Change Text Font Dialog

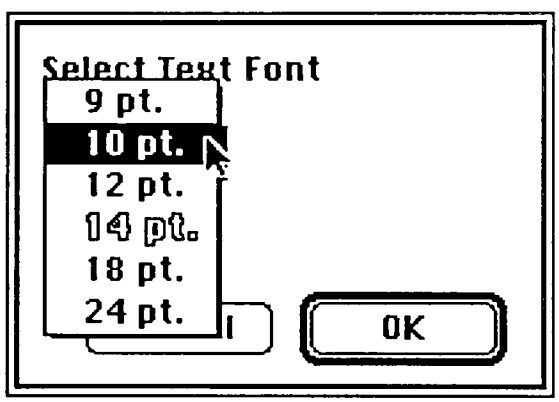


Figure B-46b -- Point. Size

Any new text objects created will be created under the current font. Previous text objects will retain their old fonts. It is not possible to combine fonts within one text object.

Change List Font/Redraw Screen

These commands have no relevance in this application.

Stack Windows

This command will cause the windows to be arranged so that their label areas are tiled. Figure B-37 shows two stacked windows.

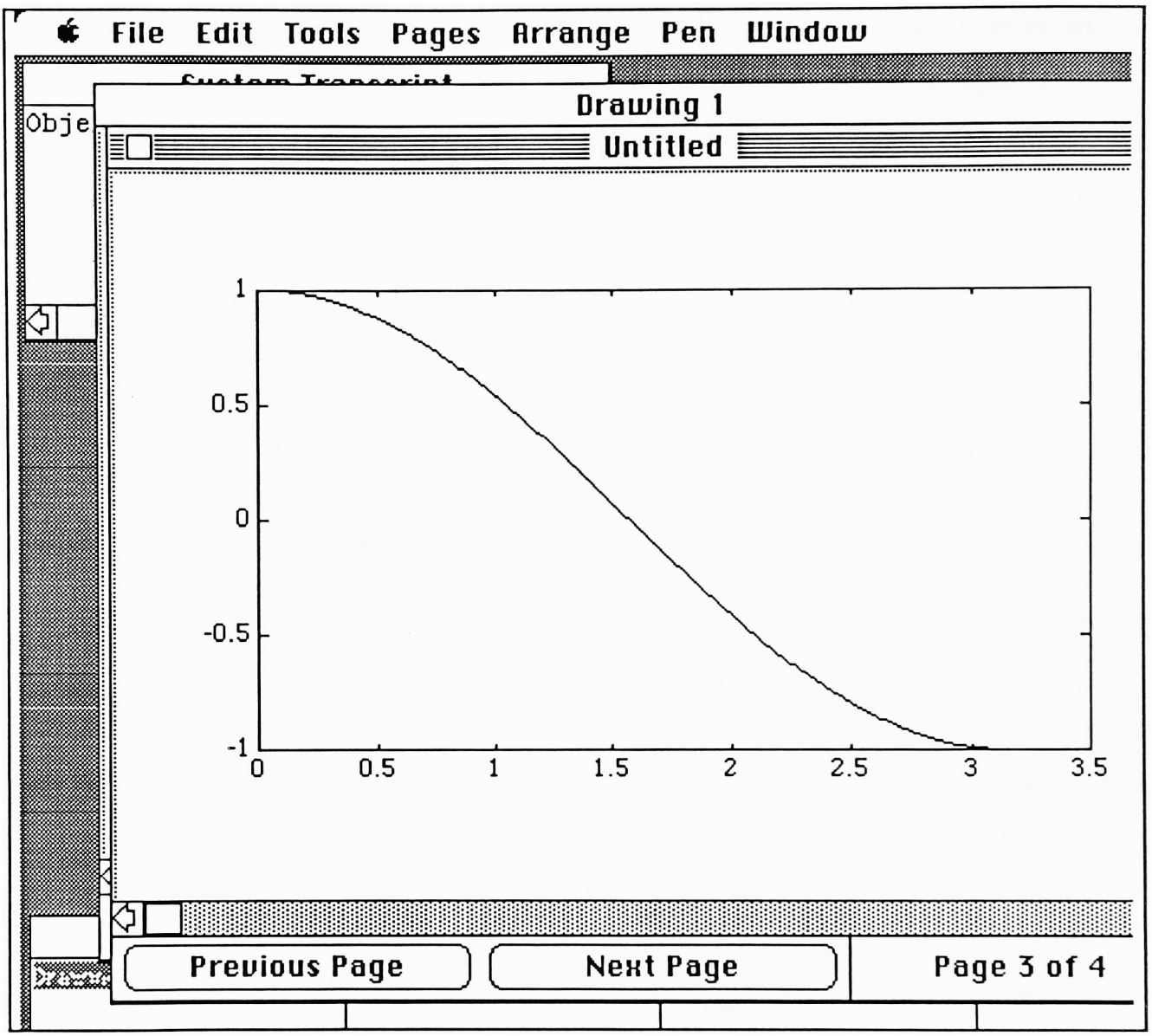


Figure B-47 -- Stacked Windows